# D-ASAP

2nd Project Meeting

# Browser model & CoreASM extensions

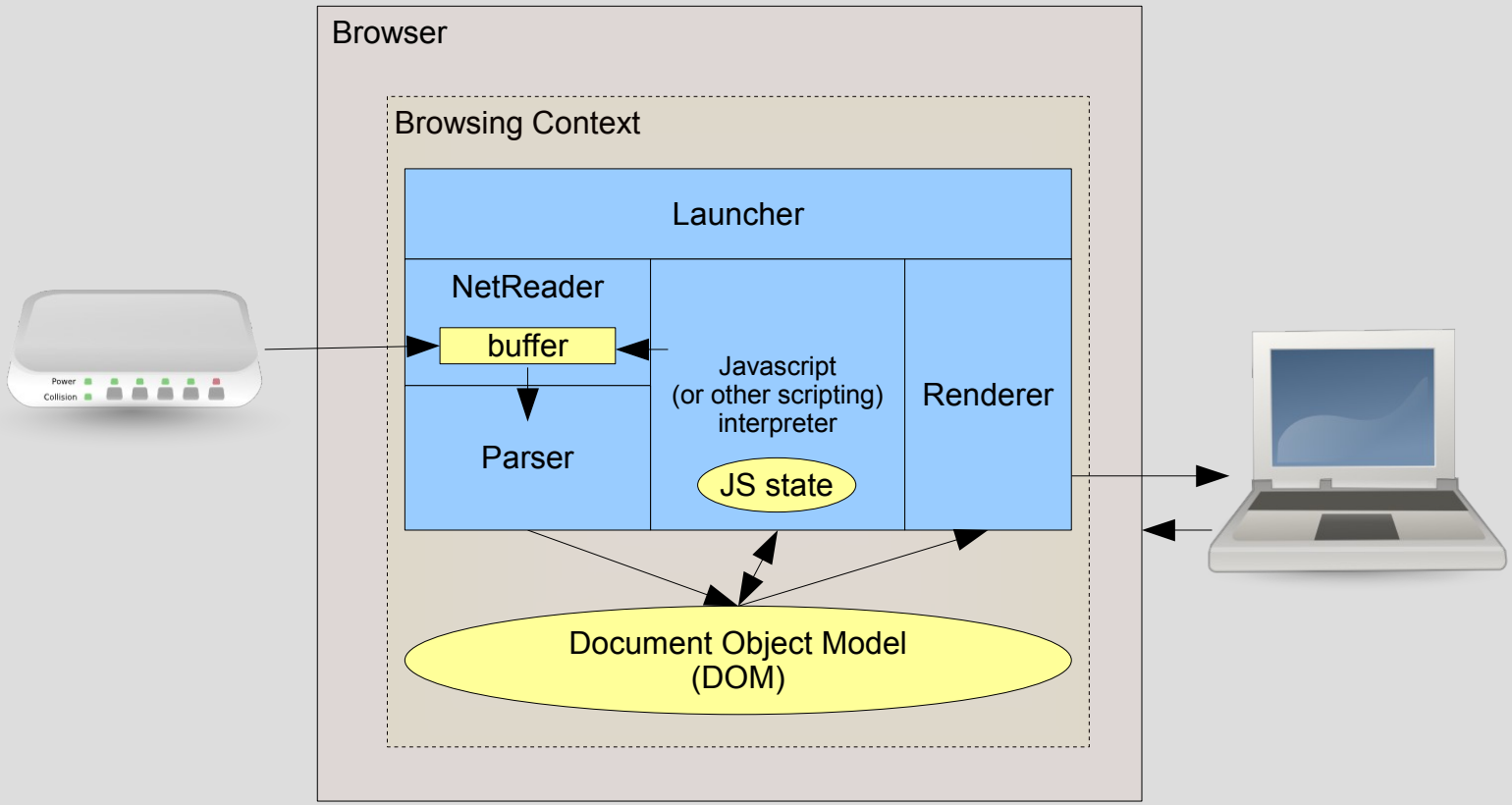
Vincenzo Gervasi
*Dipartimento di Informatica*
*Università di Pisa*

# Outline

- Browser model
  - Modeling abstract web browser behaviour
  - Use refinement to describe common browsers
- CoreASM extensions
  - Executable support for all our models
  - Need to handle hierarchical structures and languages
- Workplan
  - Where we are, where we are going

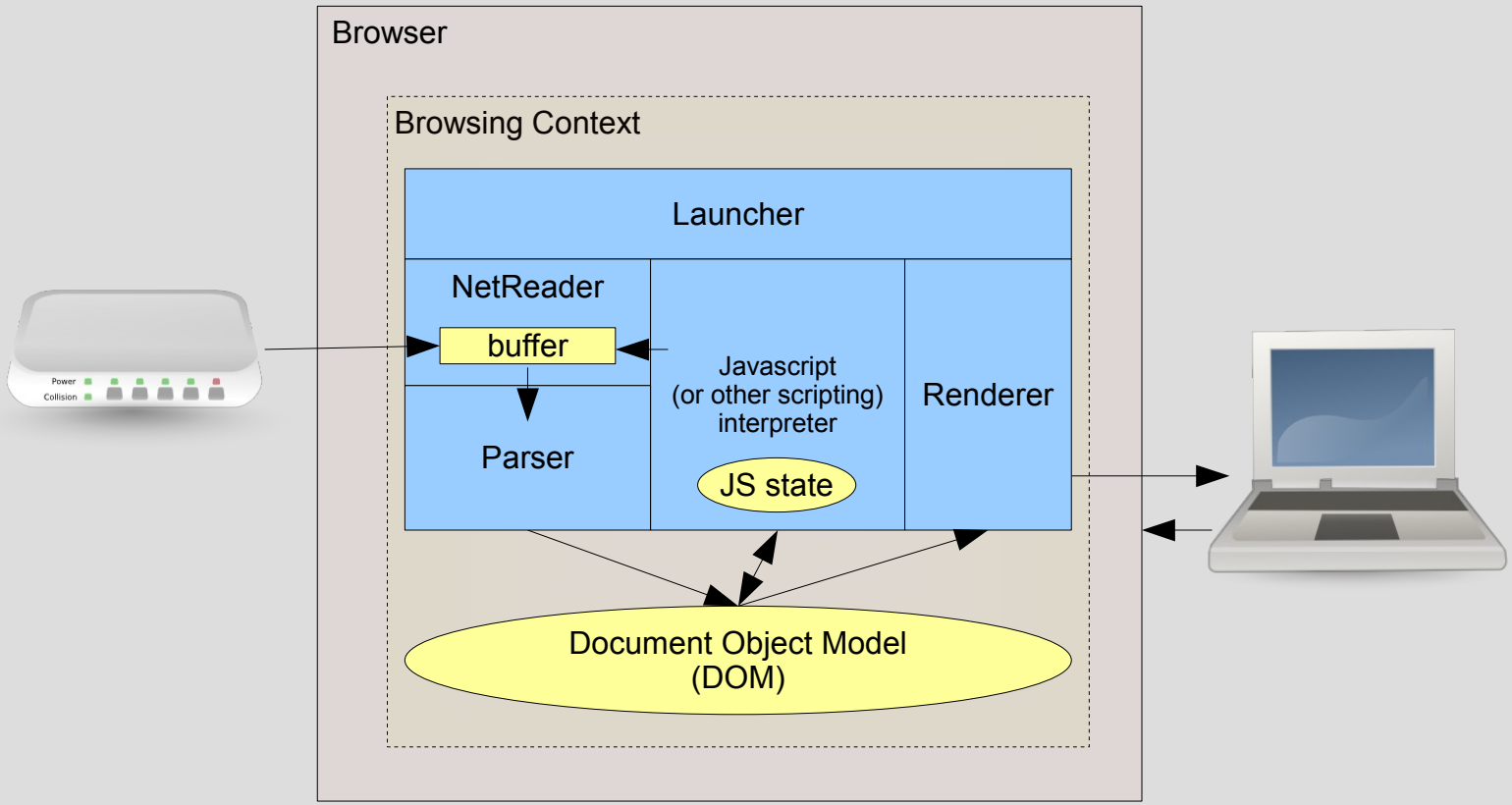
# Browser model

- A *browser* implements multiple *browsing context*
  - Each browsing context has (at least) 5 *agents*



# Browser model

- Important bits of the state (for consistency):
  - Document buffer, JS state, DOM



# Browser model properties

- Typical relevant properties:
  - ***Responsiveness***: any user action will (eventually) produce a change in the DOM
  - ***Liveness***: if a web application is currently running, it will (eventually) send requests to the server (or quit)
  - ***Consistency***: there exists an equivalence relation ≈ such that, at given *synchpoints h* and *k*, $\text{ServerSideState}_h \approx \text{BrowserSideState}_k$
  - ***Security***: insulation of BrowsingContexts, ambients, agents

# Browser model refinements

- Different browsers differ in their handling of:
  - Browsing contexts
    - Browser instances; windows; tabs; frames
  - Assignment of multiple agents (and ambients), e.g.
    - Google Chrome
      - separate processes per BC; site-based affinities
      - Multiple threads per process
    - Internet Explorer
      - 1 process per tab + 1 renderer per window
      - Multiple threads per BC (based on frames)
    - Firefox
      - Single process handles all

# Browsing context

- We focus on modeling a single browsing context

```
CreateBrowsingContext =
  CreateInitialState
  step
    LaunchRenderer
    LaunchScriptInterpreter
    if requestedLoc ≠ currLoc then
      currLoc := requestedLoc
      let b = new(Buffer)
        LaunchNetReader(requestedLoc, b)
        LaunchParser(b)
```

- We use the *ambient* construct to insulate different browsing contexts
- The context signature includes a single DOM and JS state
- Potentially multiple buffers (due to parallel loading)
- The LAUNCH* macros instantiate a new agent to start asynchronous execution
- Handling of the page is then performed by the agents

# Renderer

$$\begin{array}{l}\textsc{Renderer} = \\ \quad \textbf{if } \mathit{RenderingTime} \textbf{ and not } \mathit{Locked}(\mathit{DOM}) \\ \quad\quad \textbf{then } \textsc{GenerateUi}(\mathit{DOM})\end{array}$$

- Updating the visual can be done only at certain times
  - Opera: halfway during a computation
  - Most others: only inbetween JS computations started by different events
    - IE, Firefox: will forcibly update after a given time
- Moreover, it can only be performed if the DOM is not being modified by Parser or Interpreter

# Interpreter

```
INTERPRETER(node, program) =
  if mode = initial then
    let p = PARSE(program)
    LOCKGLOBAL(DOM)
    pos := p
    mode := interprete
  if mode = interprete then
    Perform a step of the interpreter
    if pos =undef then mode := final
  if mode = final then
    UNLOCKGLOBAL(DOM)
    mode := idle
```

- Each run of the JS interpreter describes a full computation
- Given a DOM node (e.g., for on*event*=".." code) and a program text
  - Parse the text
  - Lock the DOM
  - Execute the code
  - Unlock the DOM
- Actual execution described in Cristian's presentation
- Parsing described later

# NetReader and Parser

- NetReader
  - Requests data from network
  - Receives responses
  - If buffer unlocked, streams into buffer
  - On completion of a node, let Parser do the parsing
- May also read non-HTML data (images, CSS, etc.)

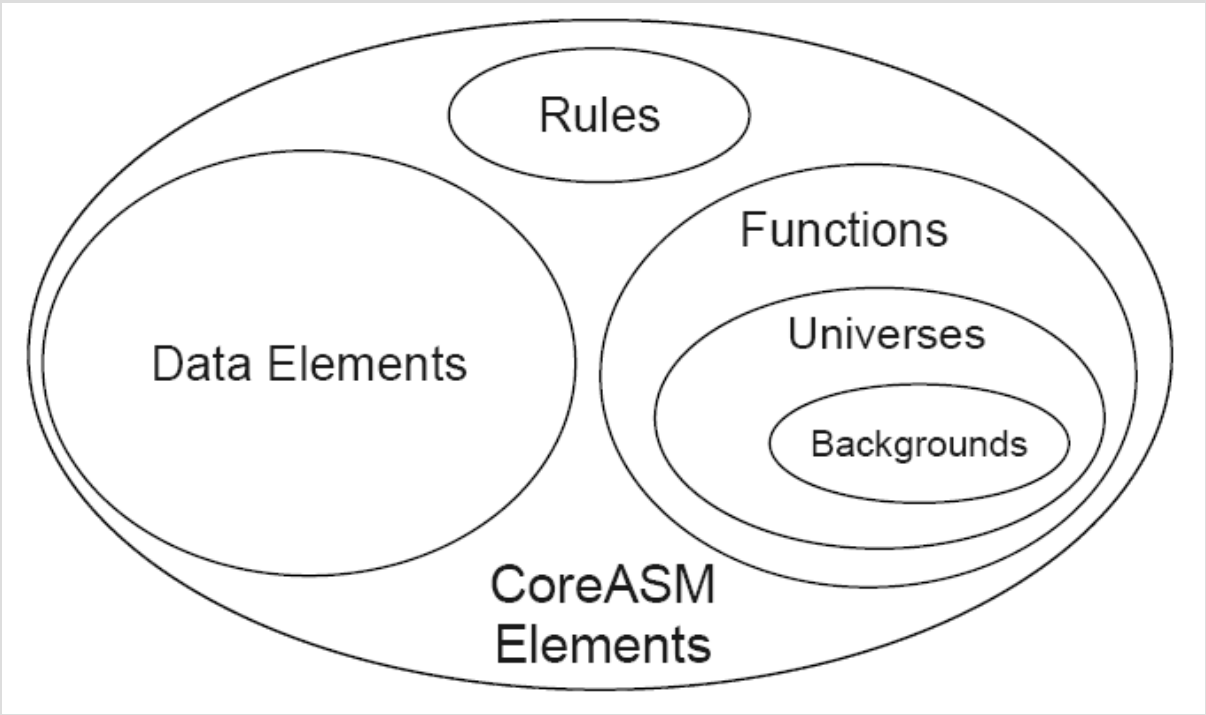
- Parser
  - Wait for signal from NetReader that there is data available
  - Parse data, build DOM nodes
  - If node is <SCRIPT>, lock buffer and run script – once finished, unlock buffer so NetReader can continue streaming
  - Interactions with BrowsingContext
    - e.g. META REFRESH

# CoreASM and executability

- We intend to have **executable** formal models
- Specification language of choice: **CoreASM**
- CoreASM has all needed features, but for two key areas:
  - A tree datatype
    - Needed for HTML, DOM, Javascript parse tree, GUI layout (possibly)
  - Parsing facilities
    - Needed to parse HTML documents, XML RPC messages, JSON, Javascript source code

# CoreASM extensibility

- CoreASM is a ***microkernel*** language
  - Only core concepts: rules, functions, booleans
    - Booleans needed only to express the characteristic function of the set of rules defined in a specification!

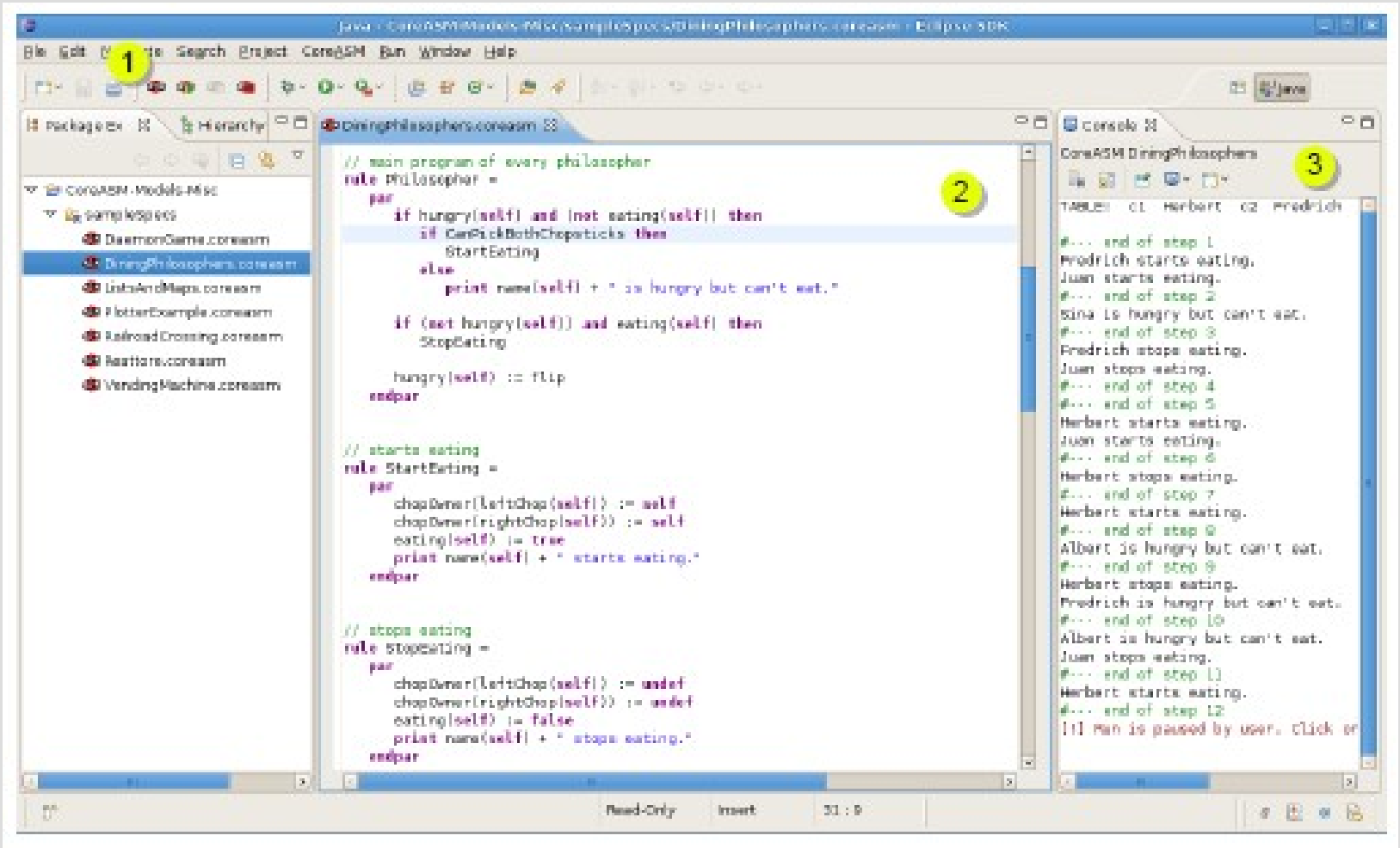


# CoreASM extensibility

- Everything else defined in plug-ins
  - Rule forms
    - Base: block, **if**, **let**, **extend**, **choose**, **forall**, **case**
    - Turbo: **seq**, **iterate**, **while**, **local**, ←, **return**
  - Primitive data types
    - Predicate logic, **exists**, **forall**, numbers and arithmetics, number ranges, strings, collections (sets, bags, lists, queues, stacks, maps)
  - Auxiliary
    - Signature, scheduling, I/O, traces, Math, time, Java bridge

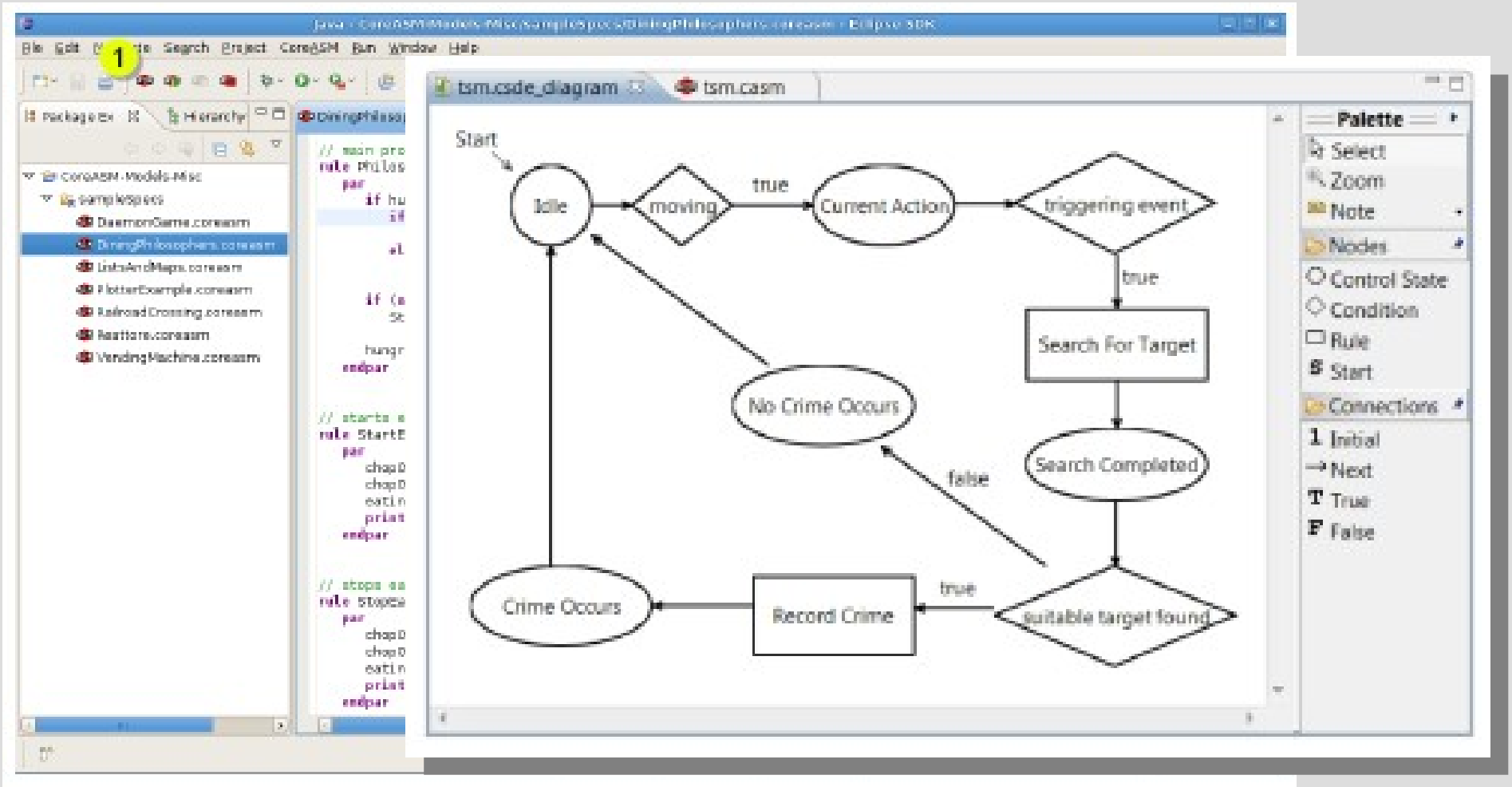
# CoreASM environment

- Rich UI embedded in Eclipse



# CoreASM environment

- Rich UI embedded in Eclipse

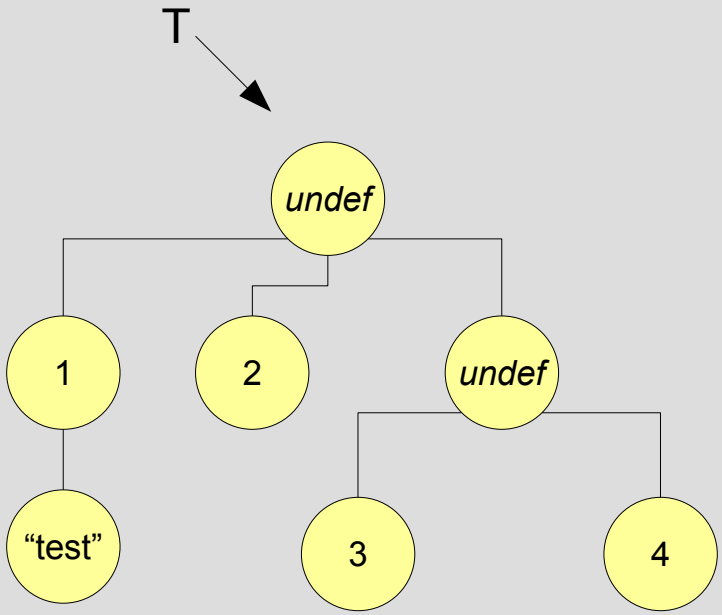


# Tree support

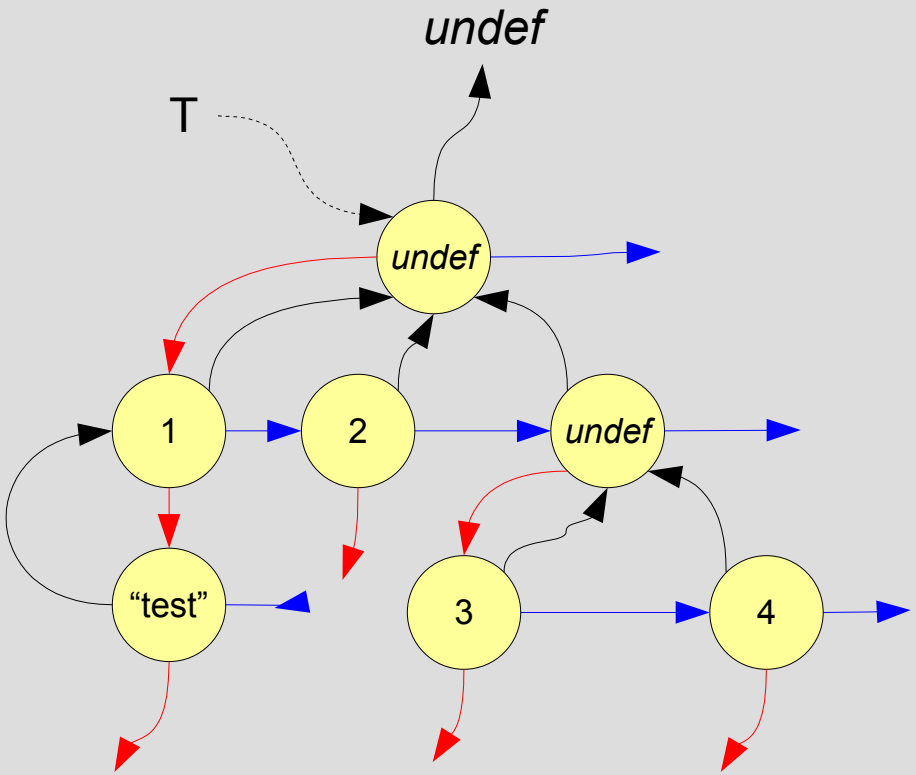
- A new domain: TREE
- Conversion to and from LIST (of lists)
- Navigation functions
  - *parent*(), *first*(), *next*()
  - rule forms for visiting (deep-first, breadth-first)
  - search functions
- Arbitrary values associated to nodes
- Updates to trees obey the usual ASM semantics
  - Defeasible state with TurboASM

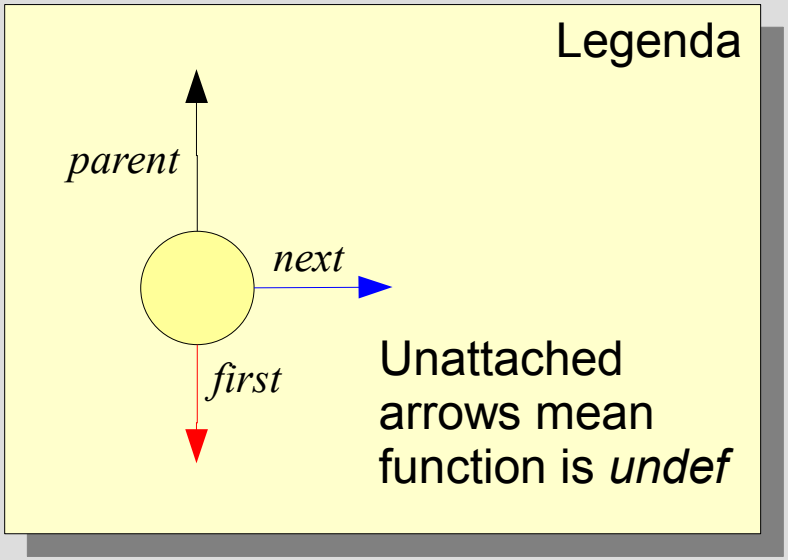
# Tree example

```
CoreASM TreeExample

use StandardPlugins
use TreePlugin

option TREE_TRAVERSAL "depth-first"

init R1

rule R1 = {
    seqblock
        make [1, 2, [3, 4]] into tree T
        add child "test" to first(T)

        v := values(T) // [undef, 1, "test", 2, undef, 3, 4]
        n := nodes(T) // [ undef , 1 , "test" , 2 , undef , 3 , 4 ]

        bft := BFT(T) // [undef, 1, 2, undef, "test", 3, 4]
        Q := next(first(T)) //  2

        print T
```



# Tree example




- Operations on trees produce a (potentially large) set of updates on these functions
- Usual semantics on conflicting updates

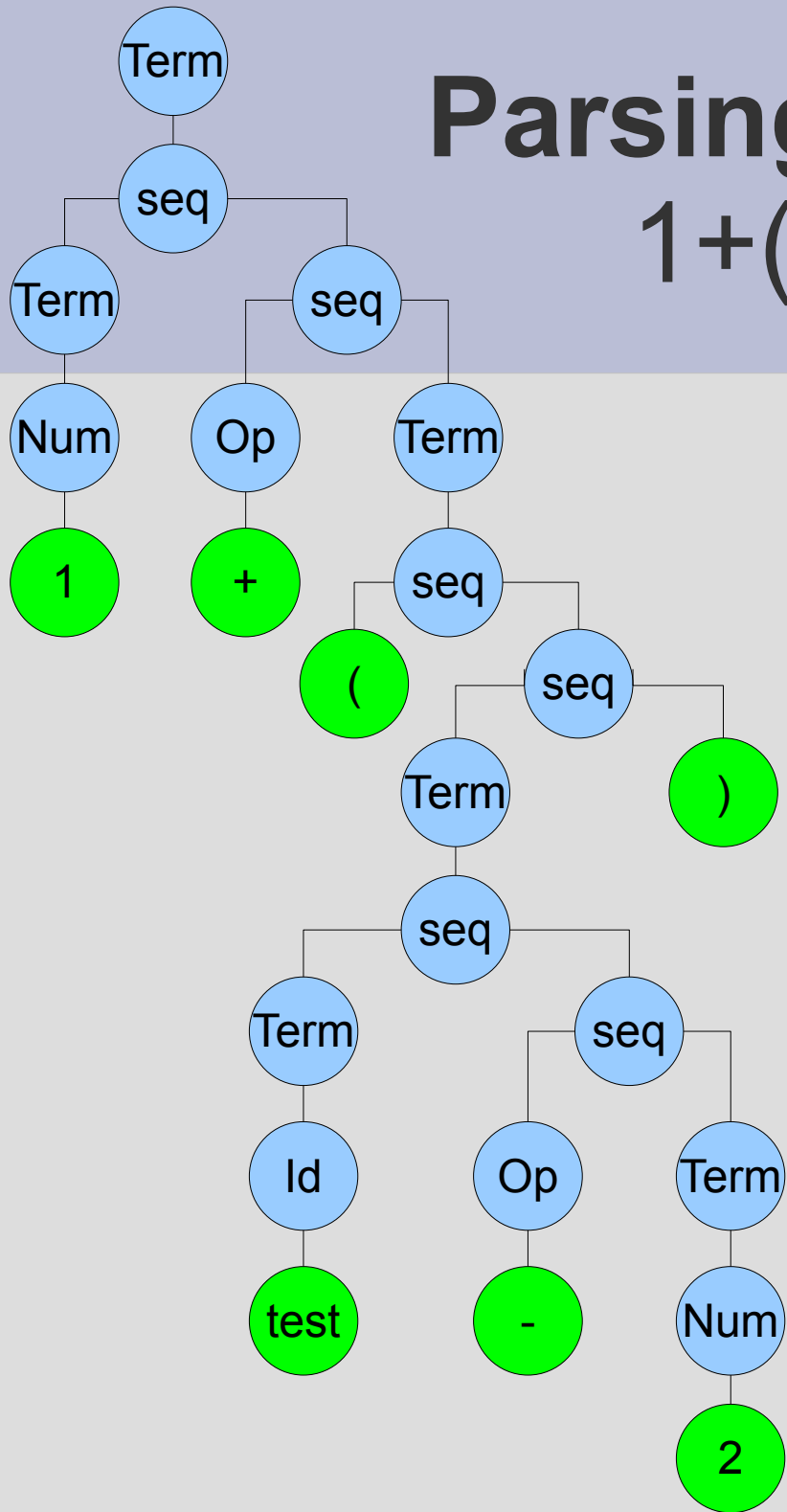
# Parsing support

- A new domain GRAMMAR
  - Grammar elements are *values* in the language
- Operators on grammars
  - Sequence, alternative, lazy evaluation, repetition
  - Automatic promotion of STRINGs to terminals
- Grammars can be defined dynamically
- Rule to parse a string according to a grammar
  - Results in a TREE (concrete syntax tree)
  - We provide functions to “read” it as an AST

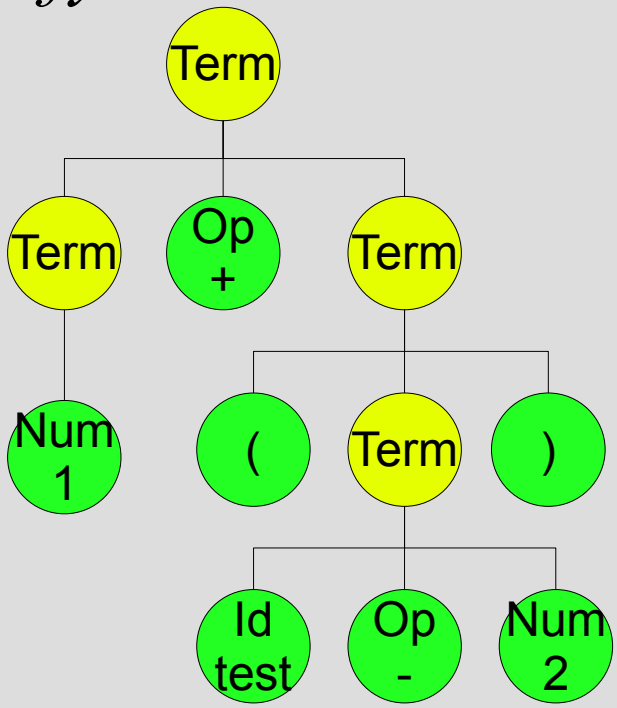
# Parsing example

```
CoreASM GrammarExample

use StandardPlugins
use TreePlugin
use GrammarPlugin

init R1

rule R1 = {
    seqblock
        Op := “+” | “-”
        Term := ID | Number | @Term . Op . @Term | “(“ . @Term . “)”

        parse “1+(test-2)” by Term into T

        print T // full concrete syntax tree

        print leaves(T) // unparse of the tree

        pos := root(T) // prepare to interpret!
```

# Parsing example
## 1+(test-2)




- Raw parse tree more complex than necessary
  - Due, e.g., to binary seq
- *simplify*: TREE → TREE



# Workplan

- On browser model
  - Detail locking mechanisms (refine by browser)
    - DOM: global locks, microlocks
    - Buffers: exact conditions
    - Handling of *asynchronous* and *deferred* execution
  - Exact handling of UI events (refine by browser)
    - Queueing policies, dispatching
    - Refresh policies, breaking locks
- On executable support
  - Finalize CoreASM extensions
  - Write HTML and Javascript grammars