# State Machines and Statecharts
# Part III: Advanced Topics


**Bruce Powel Douglass, Ph.D.**

***Chief Evangelist, i-Logix***

One man's "Of Course!" is another man's "Huh?"
*Book of Douglass, Law 79*
**Feel free to ask questions!**

**So clearly, the Boolean least squares approach allows for heterogeneous diversity when polymorphic attenuation is required.**

# A more formal definition of finite state machines

# Finite State Machine

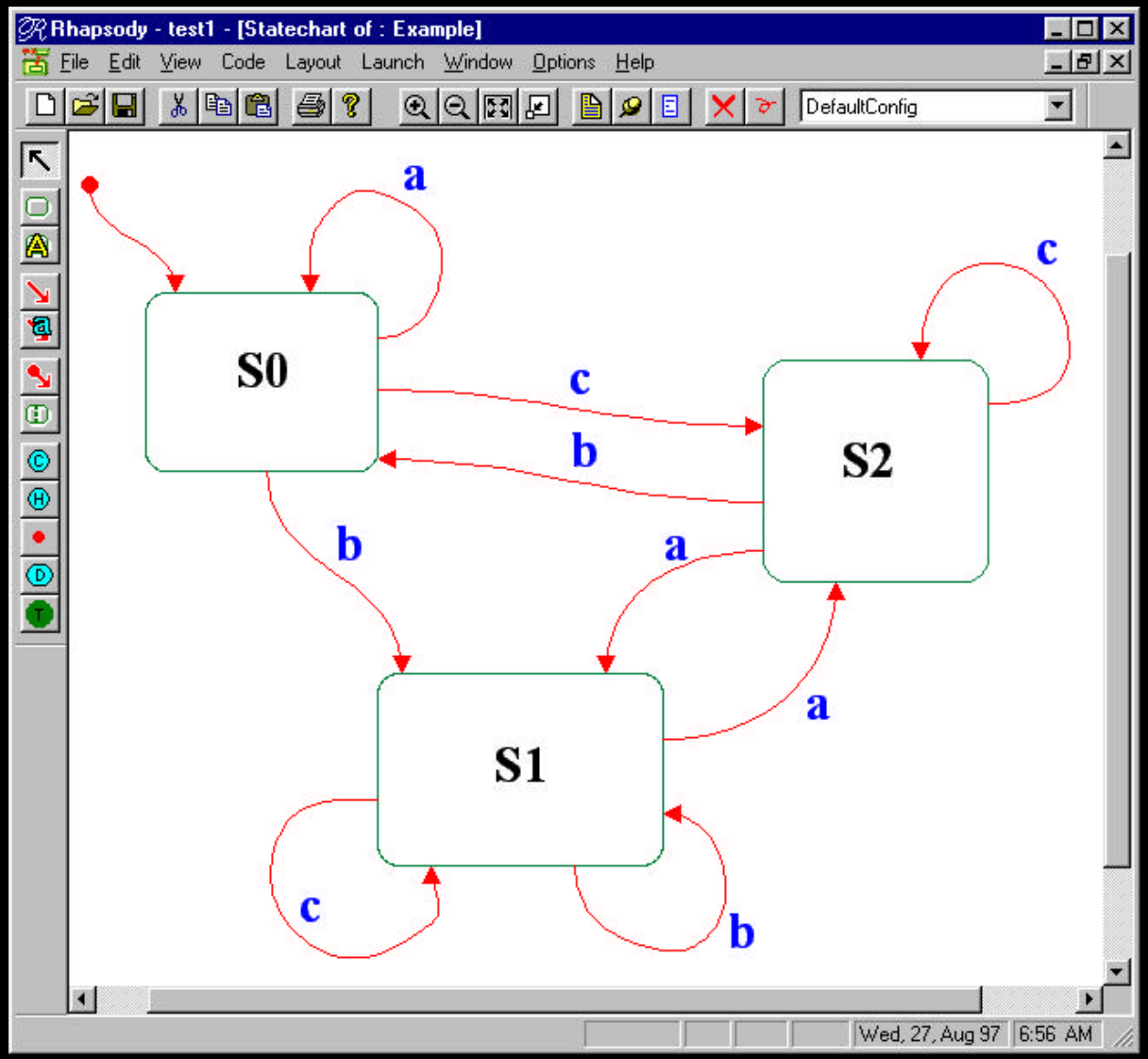
- A finite state machine (M) is an abstract model consisting of
  - a finite state set (S),
  - a starting state ($s_0$)
  - an input alphabet ($\Sigma$)
  - a mapping function $\delta = \Sigma \times S \rightarrow S$
  - an input sequence acceptance function $\beta = S \rightarrow \{0, 1\}$ such that

$$M = (S, \Sigma, s_0, \delta, \beta)$$

# Example FSM

- $S = \{s_0, s_1, s_2\}$
- $\Sigma = \{a, b, c\}$

| $\delta$ | a | b | c |
|---|---|---|---|
| $s_0$ | $s_0$ | $s_1$ | $s_2$ |
| $s_1$ | $s_2$ | $s_1$ | $s_1$ |
| $s_2$ | $s_1$ | $s_0$ | $s_2$ |



# Example 2: Parse Identifiers

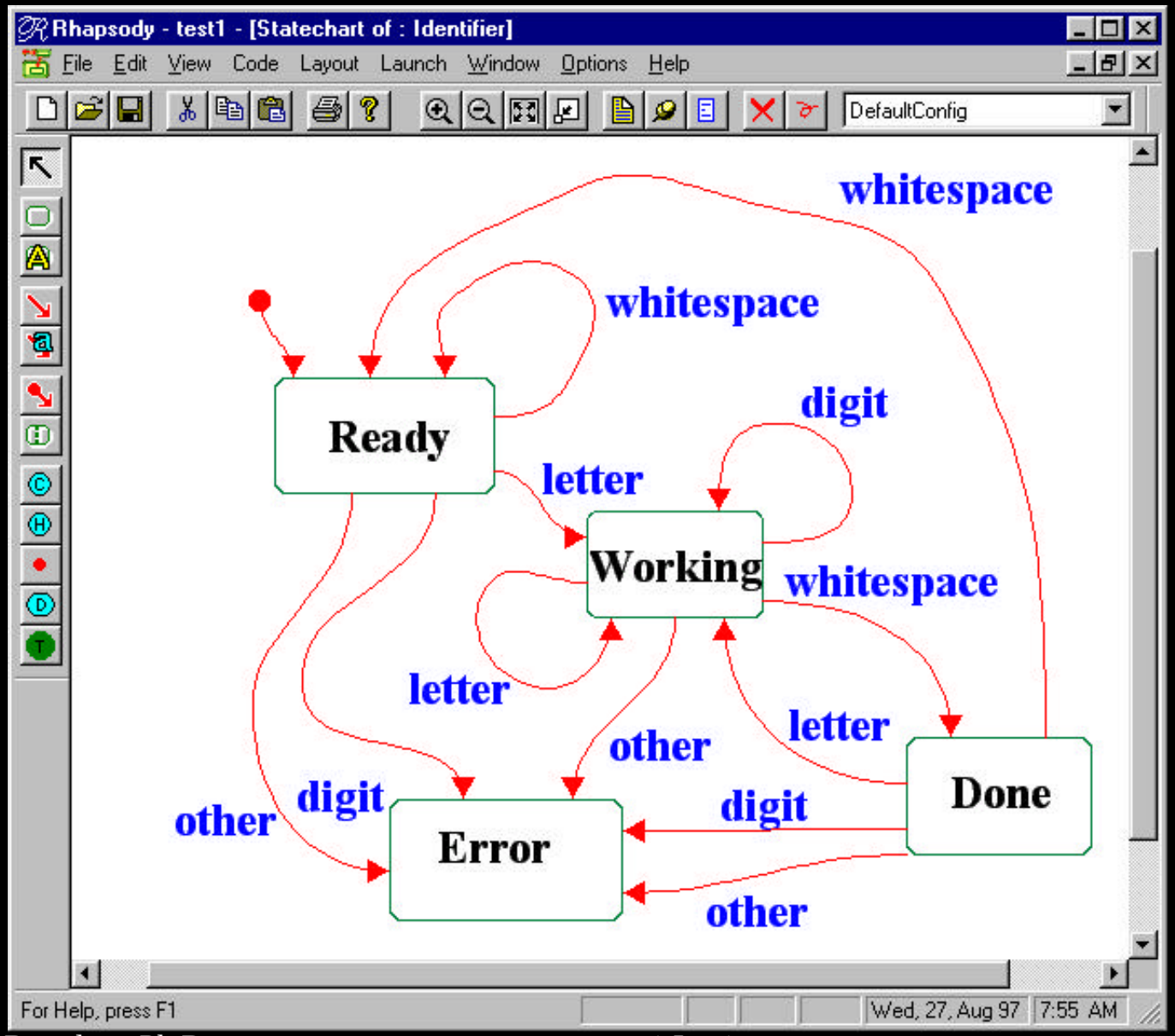
- An identifier is a sequence of characters that
  - must begin with a letter
  - may subsequently may contain any combination of letters and digits
  - is terminated by a white space character

* Example from *Fundamental Concepts of Computer Science* by Leon Levy, Dorset House Pub., 1988

# Example 2: Parse Identifiers

- S = {$s_{ready}$, $s_{working}$, $s_{done}$, $s_{error}$}
- Σ = {letter, digit, whitespace, other}
  - letter = {‘a’..’z’, ‘A’..’Z’}
  - digit = {‘0’..’9’}
  - whitespace = {‘ ‘, ‘/t’, ‘/r’, ‘/n’}
  - other = ~ { letter ∪ digit ∪ whitespace }

# Example 2: Parse Identifiers



# Example 2: Parse Identifiers

| δ | letter | digit | ws | other |
|---|---|---|---|---|
| **$S_{ready}$** | $S_{working}$ | $S_{error}$ | $S_{ready}$ | $S_{error}$ |
| **$S_{working}$** | $S_{working}$ | $S_{working}$ | $S_{done}$ | $S_{error}$ |
| **$S_{done}$** | $S_{working}$ | $S_{error}$ | $S_{ready}$ | $S_{error}$ |
| **$S_{error}$** | $S_{error}$ | $S_{error}$ | $S_{error}$ | $S_{error}$ |

# Example 3: Gnomes & Warlocks

There are 2 Gnomes and 2 Warlocks on the left bank of a river. They want to cross the river but their rowboat can only be rowed by a single person and can carry only two people. Warlocks should never outnumber Gnomes in any location if there are any Gnomes at the location because they'll kill the schmuck. Describe the various ontological states and transitions, and describe a path through the state machine that achieves the goal state (everyone on the right side of the river).

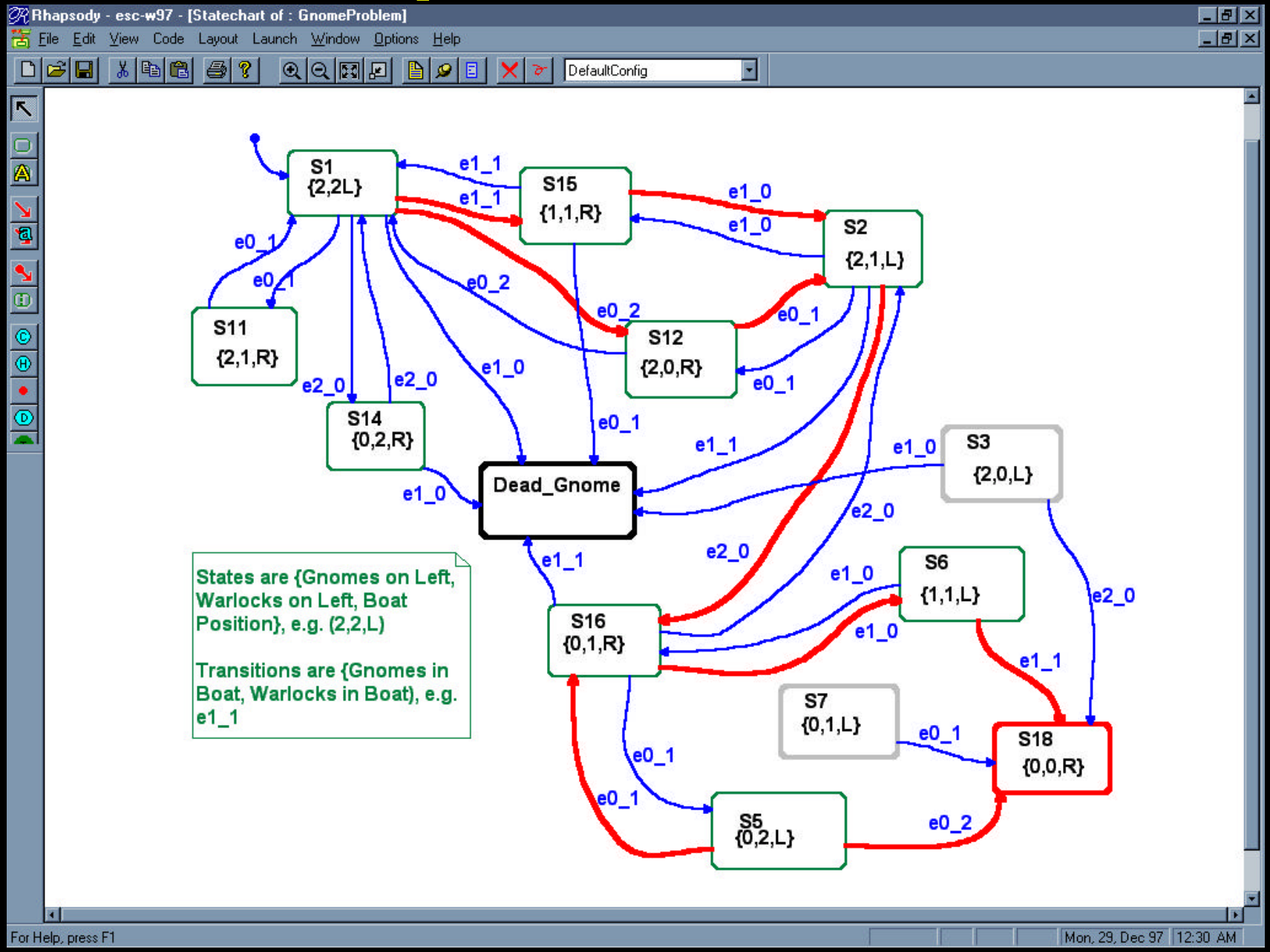
# Example 3: Gnomes & Warlocks

| S | {g,w,b} | 0,1 | 1,0 | 1,1 | 2,0 | 0,2 |
|---|---|---|---|---|---|---|
| $S_1$ | 2,2,L | $S_{11}$ | $S_{13}$ | $S_{15}$ | $S_{14}$ | $S_{12}$ |
| $S_2$ | 2,1,L | $S_{12}$ | $S_{15}$ | $S_{17}$ | $S_{16}$ | - |
| $S_3$ | 2,0,L | - | $S_{17}$ | - | $S_{18}$ | - |
| $S_4$ | 1,2,L * | Dead Gnome | | | | |
| $S_5$ | 0,2,L | $S_{16}$ | - | - | - | $S_{18}$ |
| $S_6$ | 1,1,L | $S_{17}$ | $S_{16}$ | $S_{18}$ | - | - |
| $S_7$ | 0,1,L | $S_{18}$ | - | - | - | - |
| $S_8$ | 1,0,L * | Dead Gnome | | | | |
| $S_9$ | 0,0,L ? | Impossible state | | | | |
| $S_{10}$ | 2,2,R ? | Impossible state | | | | |
| $S_{11}$ | 2,1,R | $S_1$ | - | - | - | - |
| $S_{12}$ | 2,0,R | $S_2$ | - | - | - | $S_1$ |
| $S_{13}$ | 1,2,R * | Dead Gnome | | | | |
| $S_{14}$ | 0,2,R | - | $S_4$ | - | $S_1$ | - |
| $S_{15}$ | 1,1,R | $S_4$ | $S_2$ | $S_1$ | - | - |
| $S_{16}$ | 0,1,R | $S_5$ | $S_6$ | $S_4$ | $S_2$ | - |
| $S_{17}$ | 1,0,R * | Dead Gnome | | | | |
| $S_{18}$ | 0,0,R !! | ***GOAL*** | | | | |

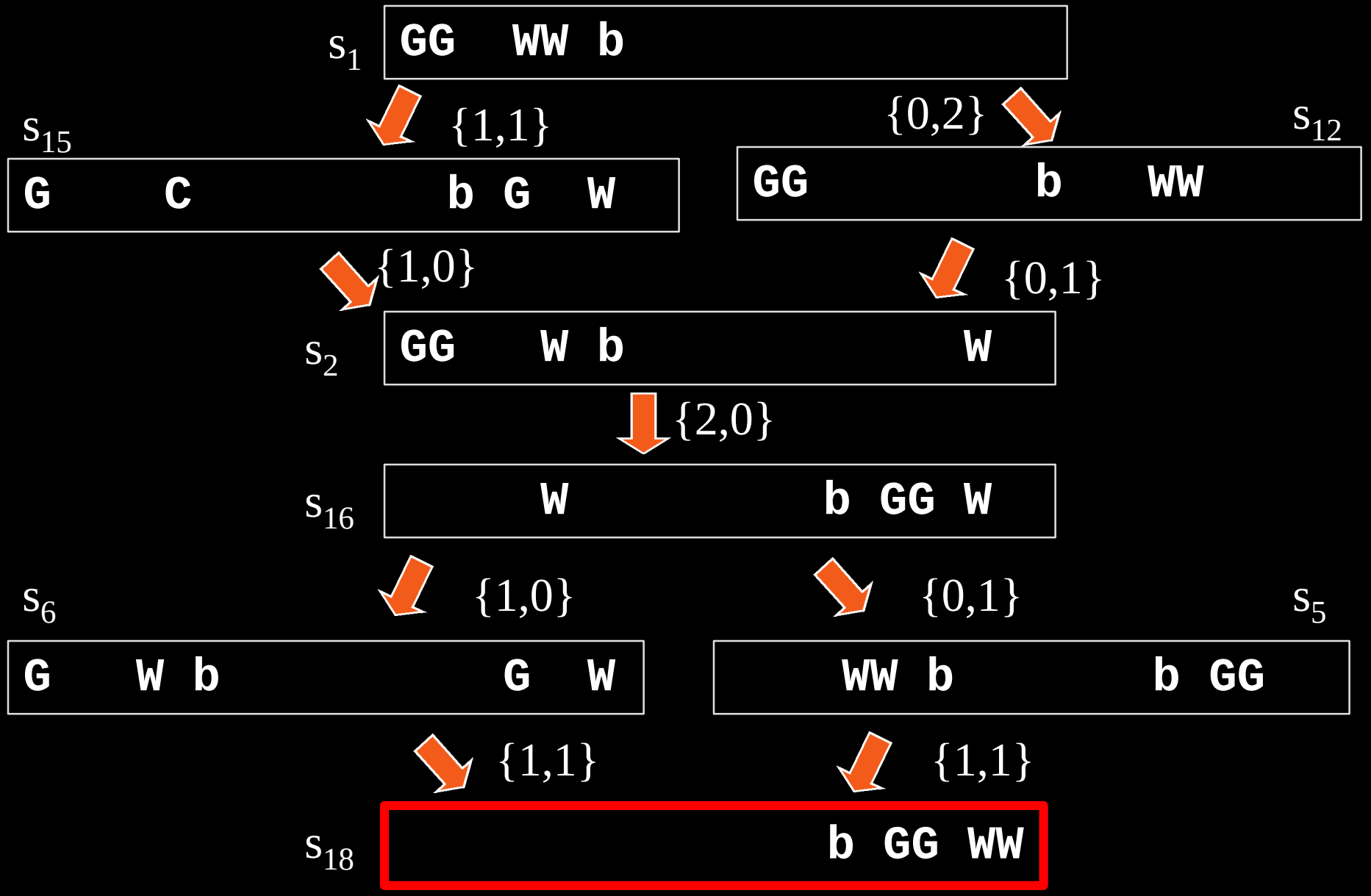
S = {Gnomes on left, Warlocks on left, position of boat}

Event = {Gnomes in boat, Warlocks in boat}

# Example 3: Gnomes & Warlocks



# Example 3: Gnomes & Warlocks

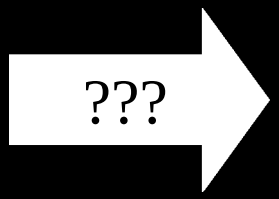


# Take Home Problem

The contents of any square may be moved to any adjacent empty square. What sequence of moves is required to get from the starting state to the goal state?

Starting State

| | | |
|---|---|---|
| A | A | B |
| A | | C |




Goal State

| | | |
|---|---|---|
| A | A | C |
| A | | B |

* Example from *How to Solve Problems: Elements of a Theory of Problems and Problem Solving* by Wayne Wickelgren, W. H. Freeman & Co., 1974

# Limits of Finite State Machines

# What are FSMs *BAD* at?

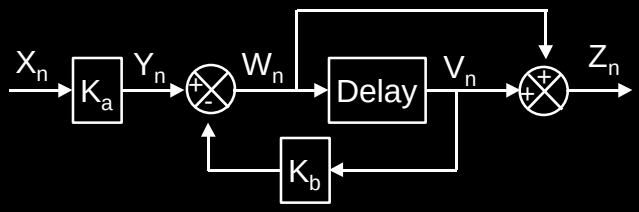
- Infinite state spaces
- Continuous systems
  - Continuous mathematics
  - Fuzzy systems
  - neural systems
- Context-free grammars

# Infinite State Spaces

- Construct a FSM for the rule
  - S = {0, 1, 2, …}
  - Σ = { '++' }
  - $s_0$ = { 0 }

# Continuous Mathematics

- Construct a FSM for to represent the control loop




- Construct a FSM for the moving average filter function

$$f_j = \frac{d_j + d_{j-1} + d_{j-2} + d_{j-3}}{4}$$

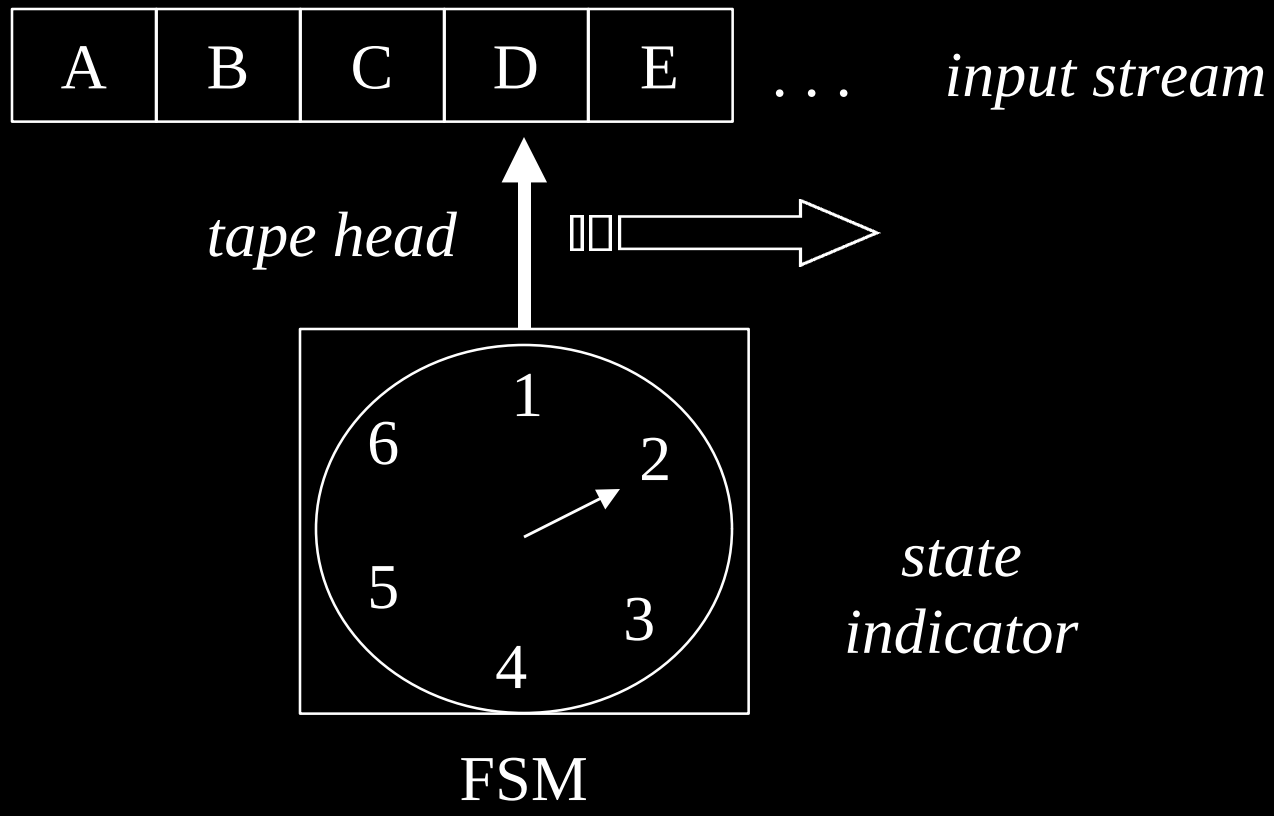
# Other kinds of machines

# Other Kinds of Machines

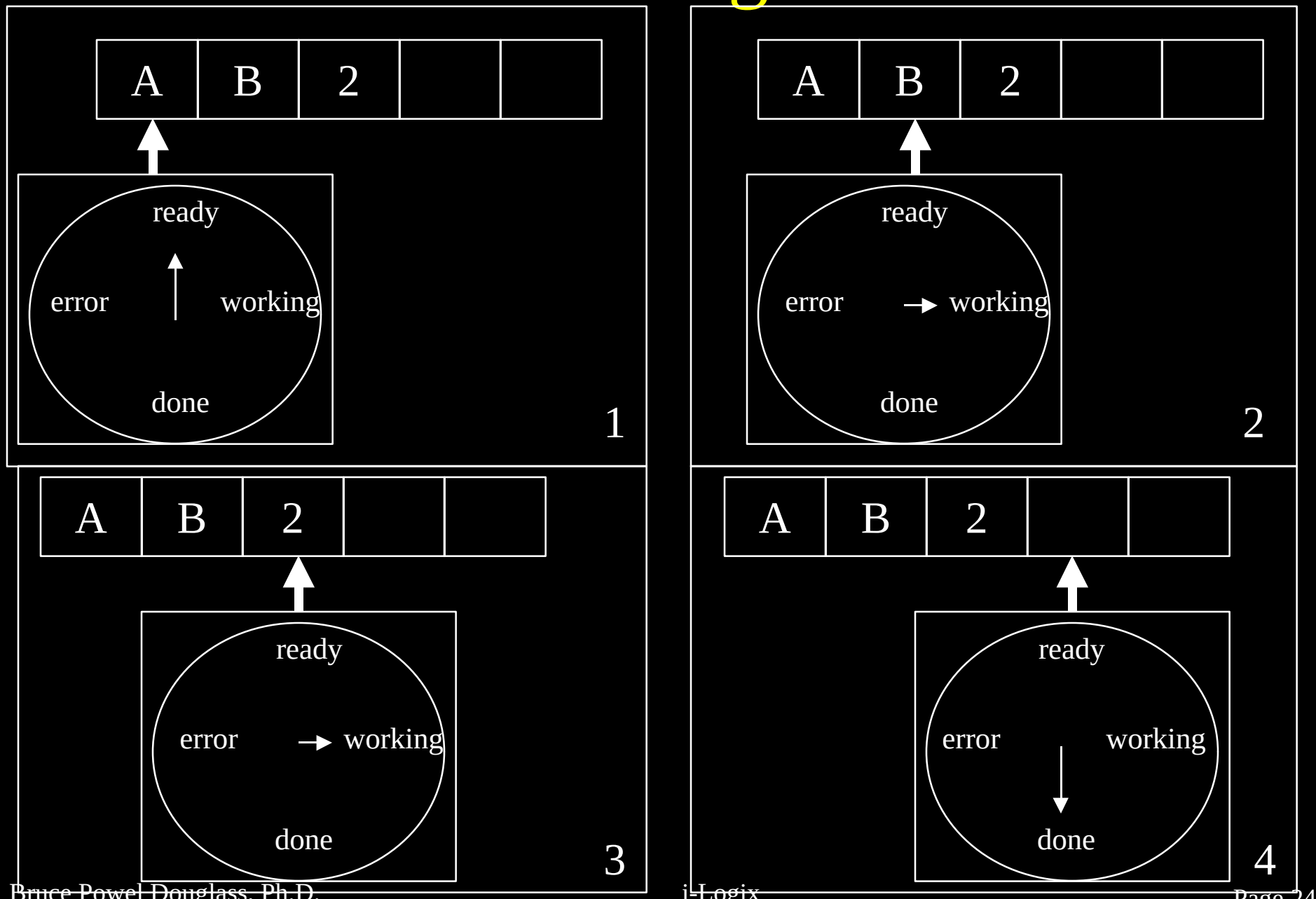
- Counting Machines
- Stack Machines

# FSM as Tape Reader

- A FSM may be thought of as a machine that reads an input stream from a tape.
  - The read values on the tape are events

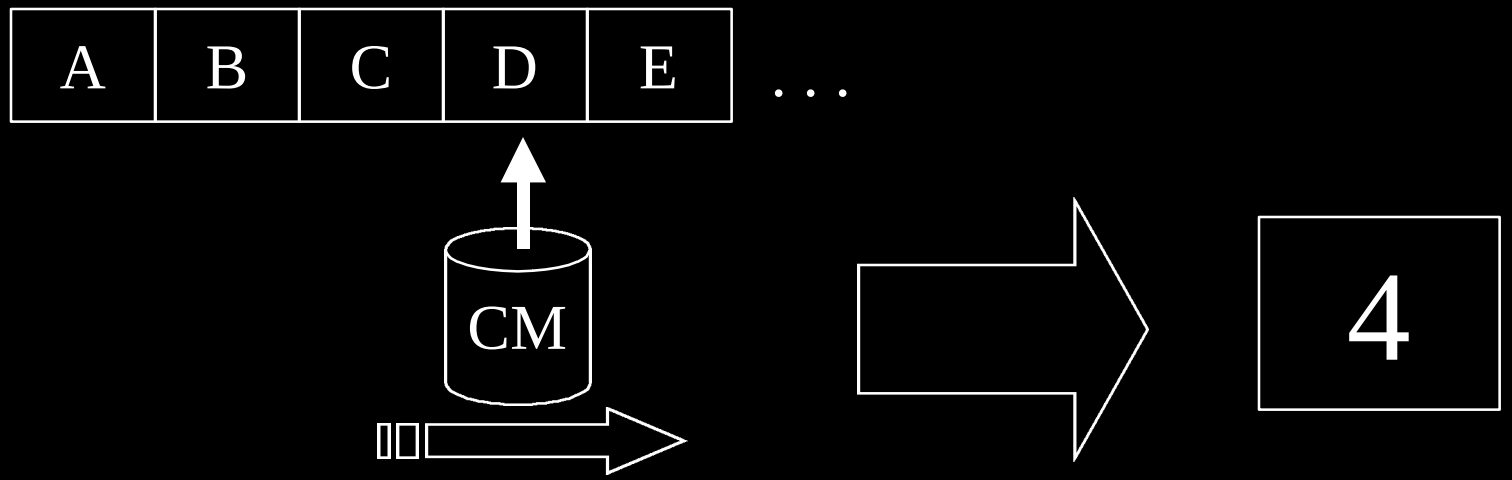


# FSM for Parsing Identifier

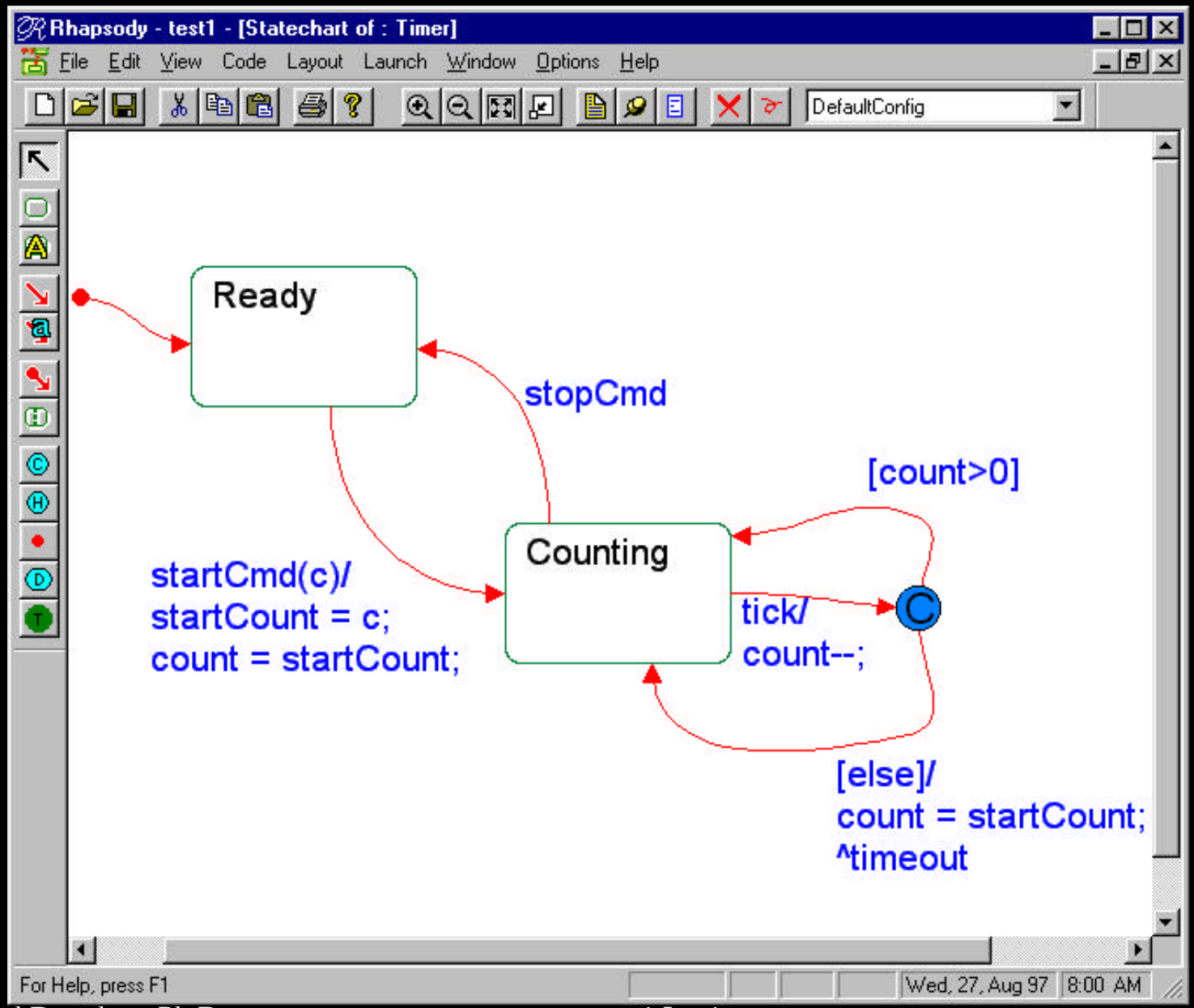


# Counting Machine

- FSMs have no memory -- they must assume a new state for each count
- A counting machine can count an arbitrarily large number of states

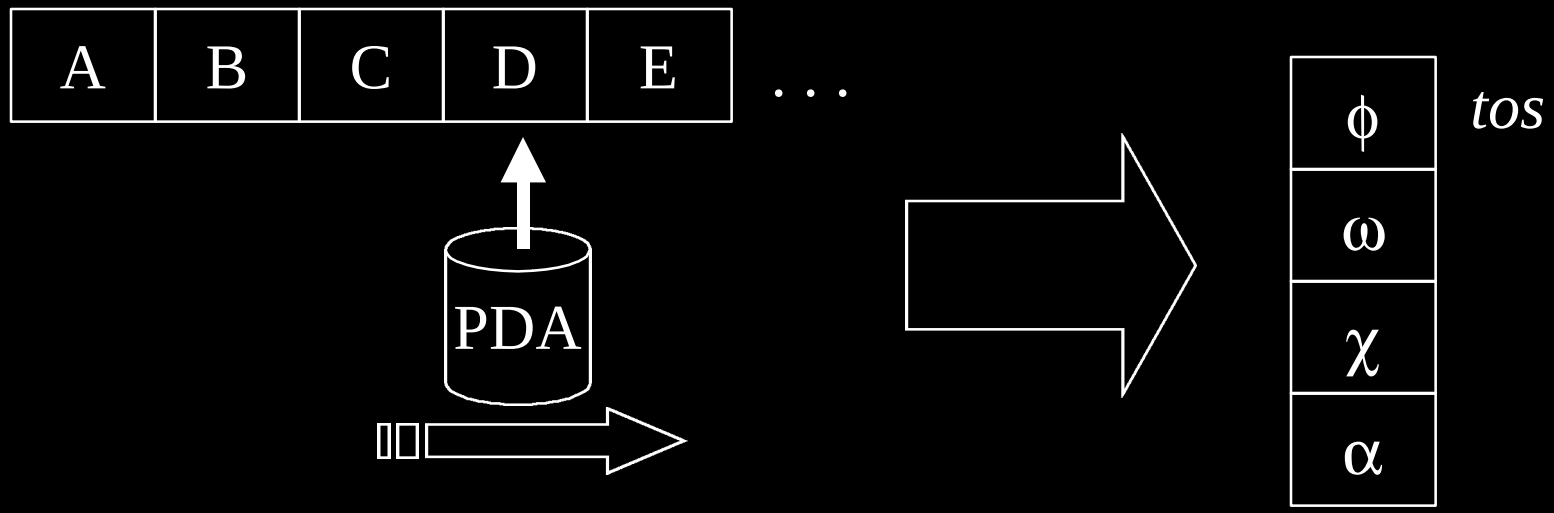


# Counting Machine Example



# Pushdown Stack Automata

- FSMs have no memory -- only ontological states
- PDAs have
  - FSM
  - Stack (may have a different vocabulary)
- PDAs can parse context-free grammars



# Types of Grammars

- Chomsky identified 4 different kinds of grammars
  - Unrestricted (UG)
  - Context-sensitive (CSG)
    - many programming languages
    - arithmetic expressions
    - Addressable by pushdown automata
  - Context-free (CFG)
  - Right-linear (RLG)
    - regular expressions
    - addressable by state machines

# Context-Free Grammars

- Productions of the form

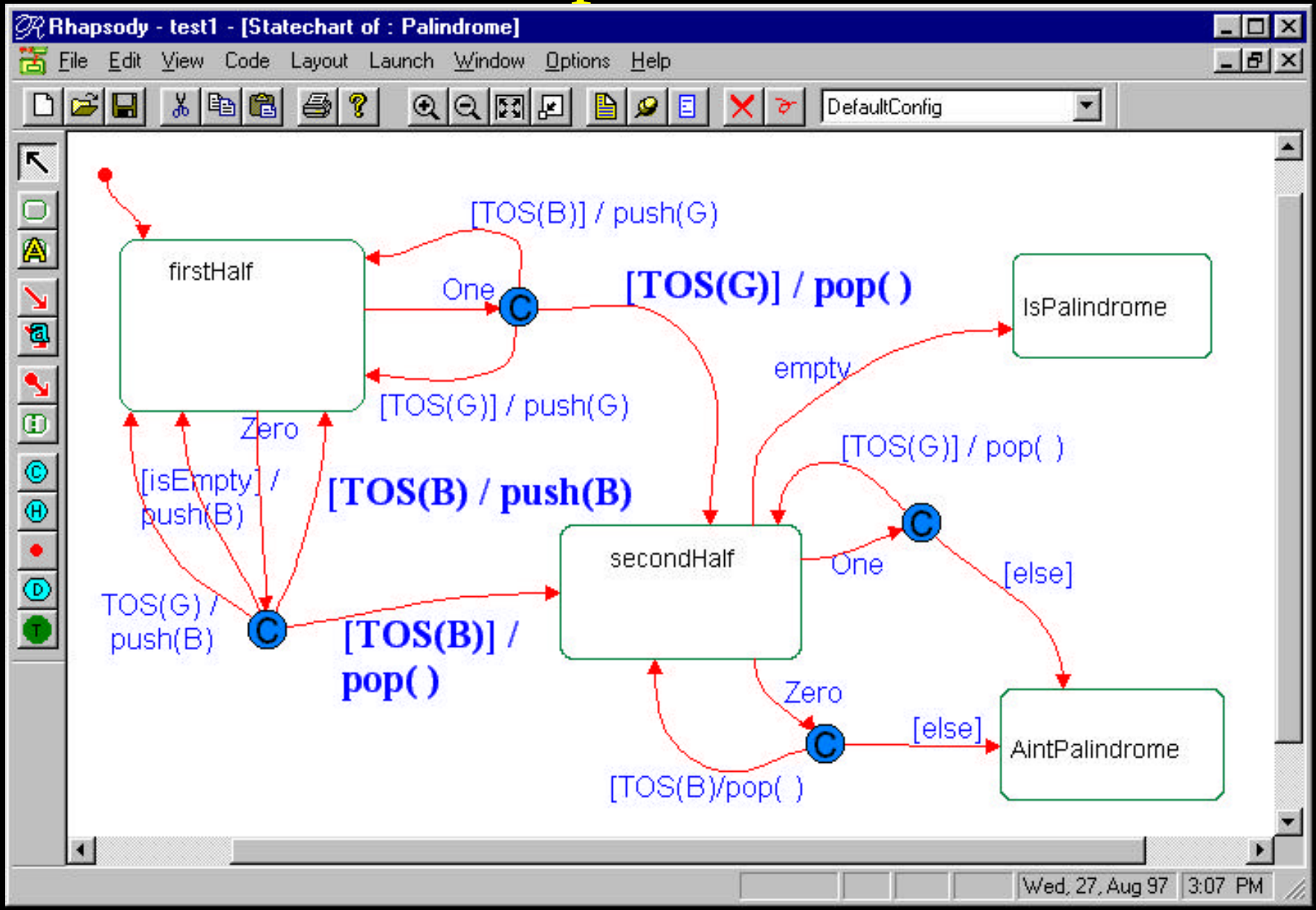
  $$P: x \rightarrow y$$

  - x is a member of N
  - y is a member of $(N \cup \Sigma)^*$
  - N is a nonterminal alphabet
  - $\Sigma$ is a terminal alphabet
  - P is a set of production rules
  - S is a goal symbol
- A PDA can recognize a CFG

# PDA Example: Palindrome

- A Palindrome is a series of symbols which is symmetric around its middle, i.e. 0110110110
- A nondeterministic PDA can be used to determine if an input string is a palindrome
  - We will consider only binary palindromes s = { 0, 1}.
  - Stack alphabet = {E (empty), B, G}
  - Stack begins empty {E}

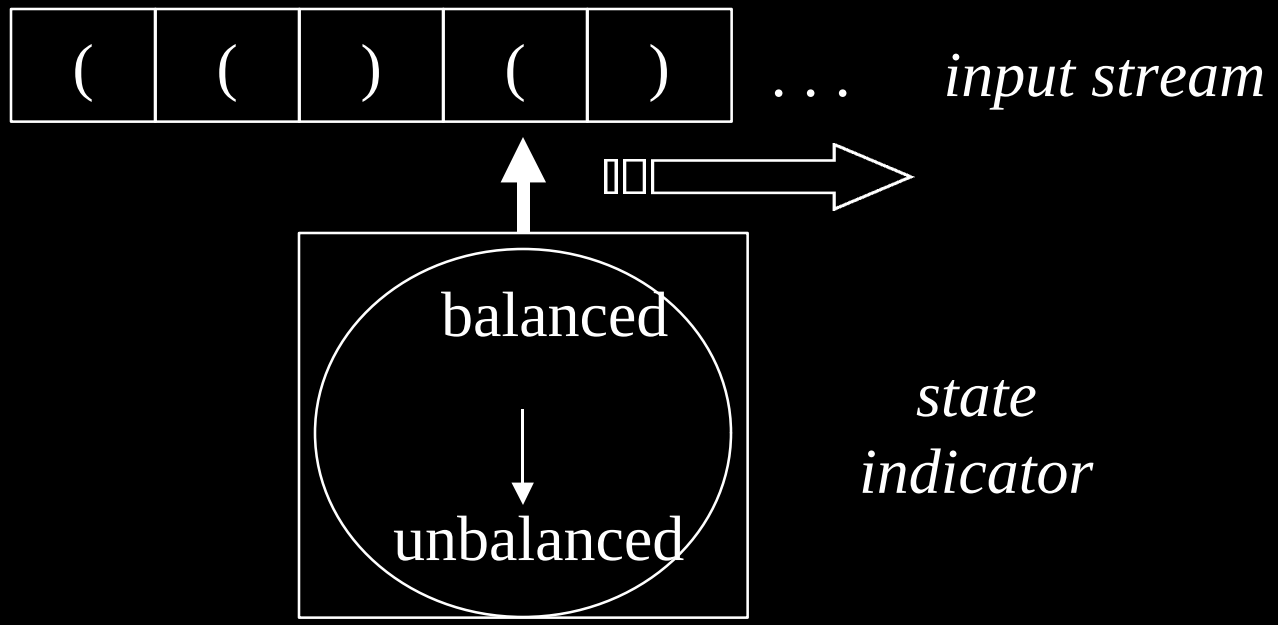
# PDA Example: Palindrome



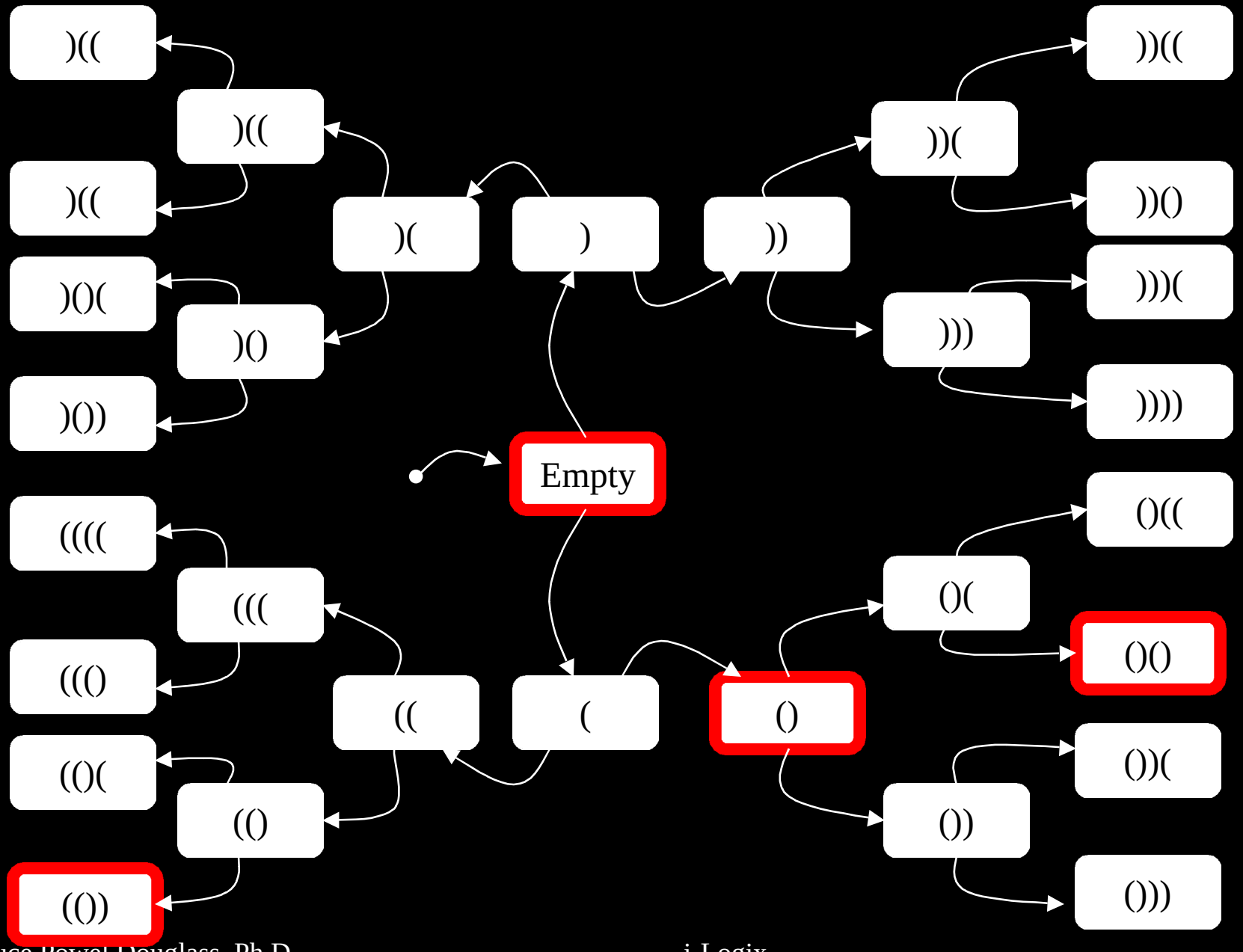
# Examples

# Balancing Parentheses

- Build a machine that can balance arbitrary parenthetical expressions
- Note that no FSM can do this because it requires an infinite set of states or memory:
  - '(' '((' '(((' '((((' are all different conditions



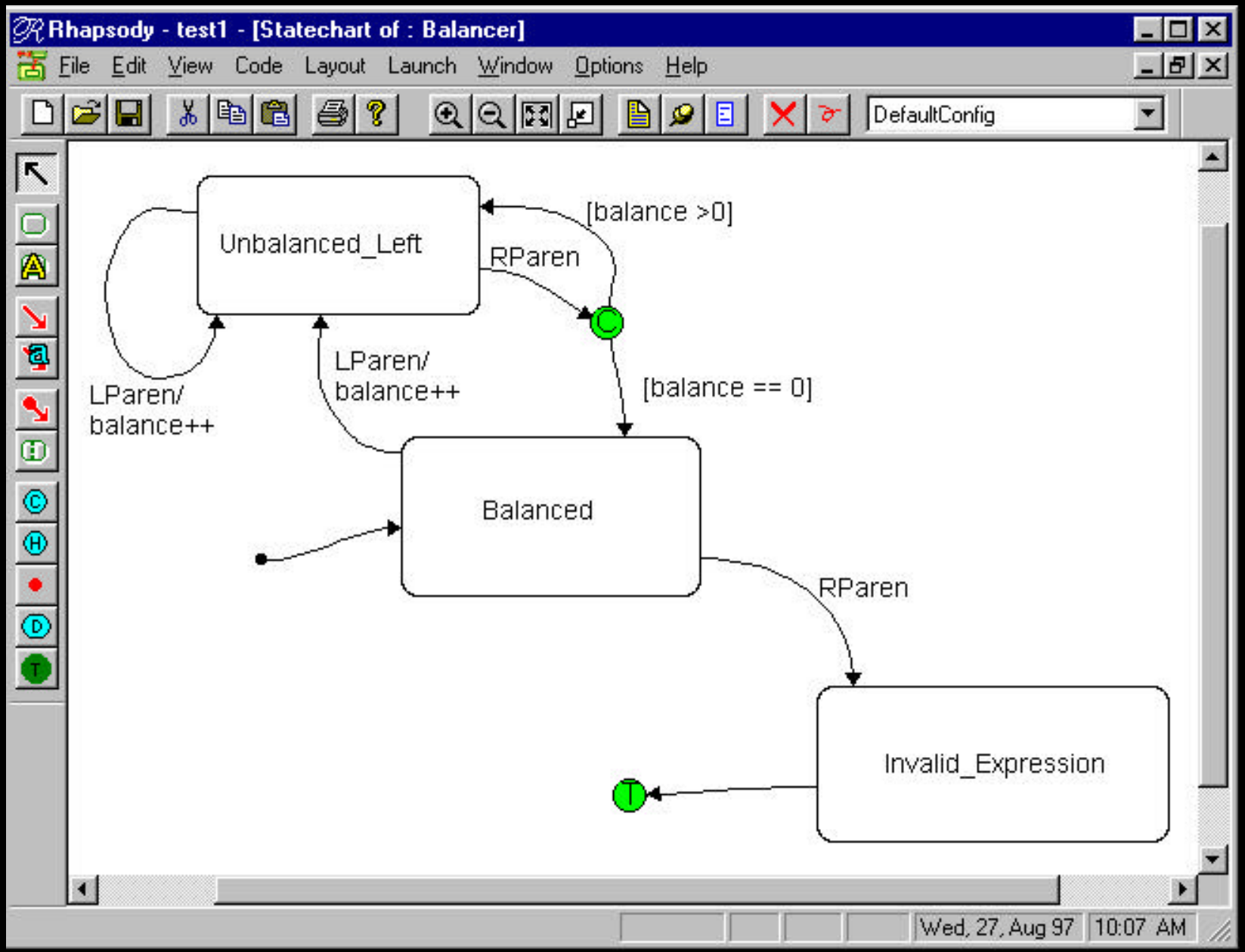
# Balancing 4 Parentheses



# Balancing Parentheses (with Memory)

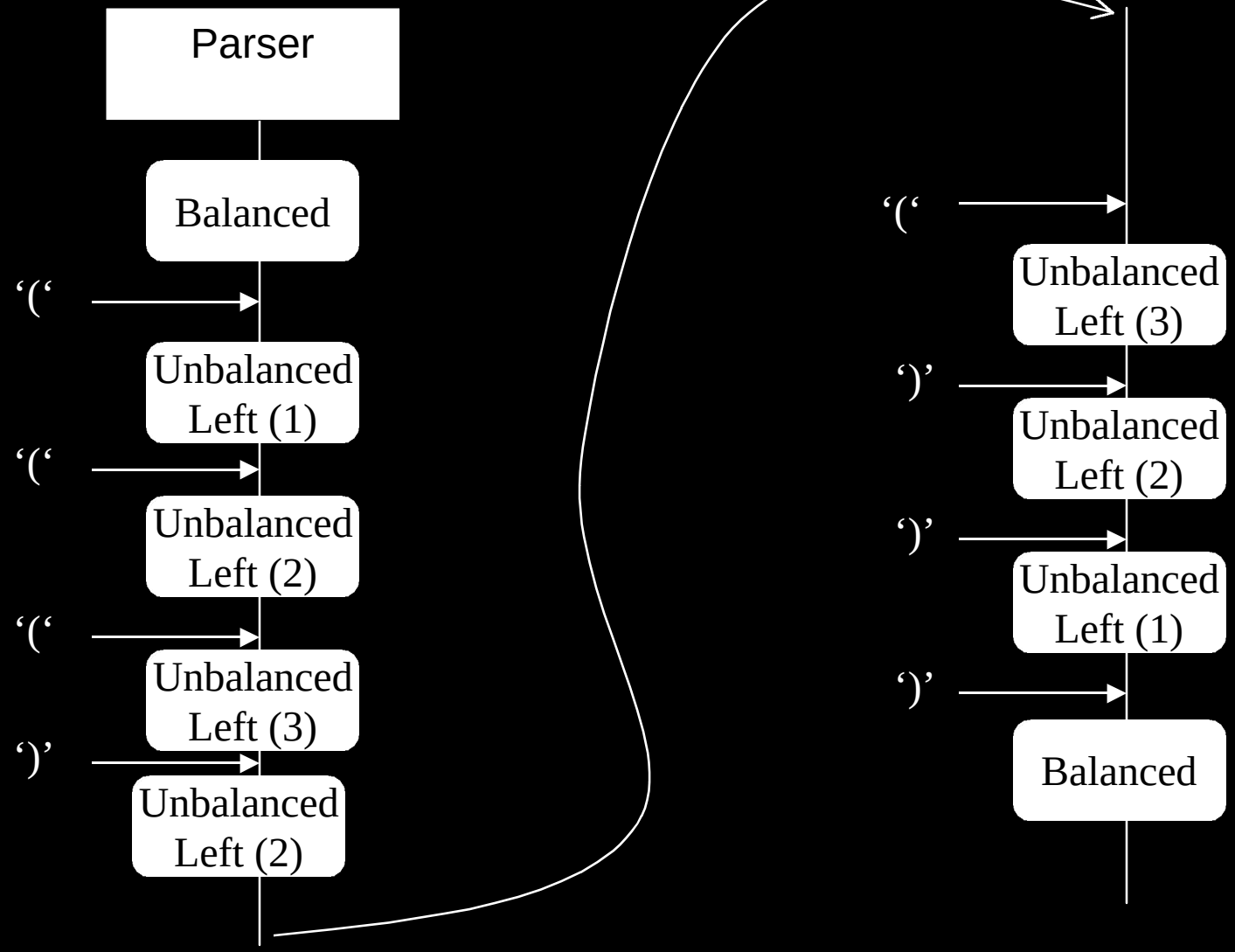
- If we add a counting machine (i.e. memory), note that state characteristics:
  - The number of characters is even (0, 2, 4, ...)
  - The number of left parentheses equals the number of right parentheses
  - The next token received when the state corresponds to a balanced expression must be a left parenthesis.

# Balancing Parentheses (with Memory)



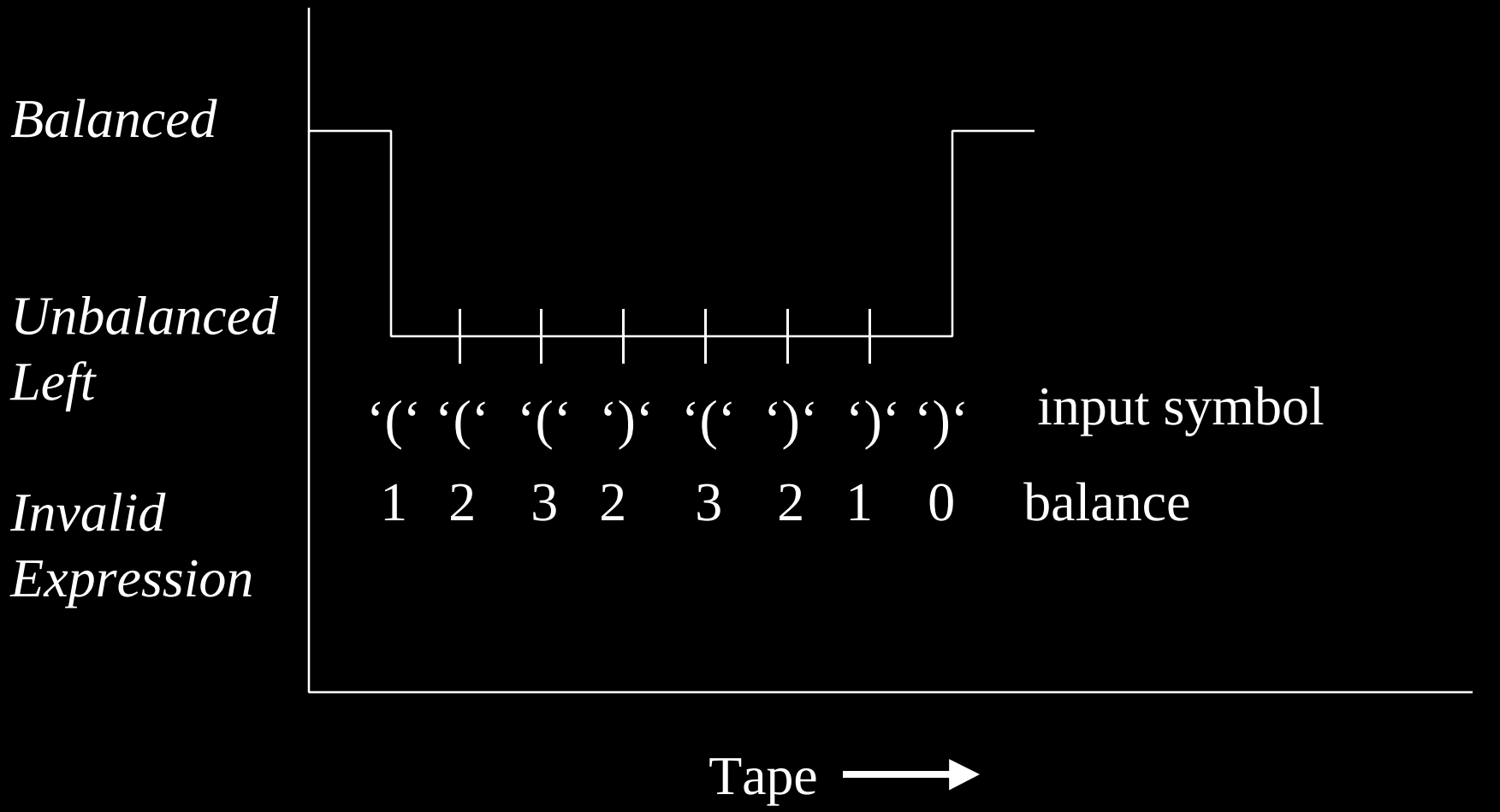
# Example

- Balance ‘( ( () () ) )’



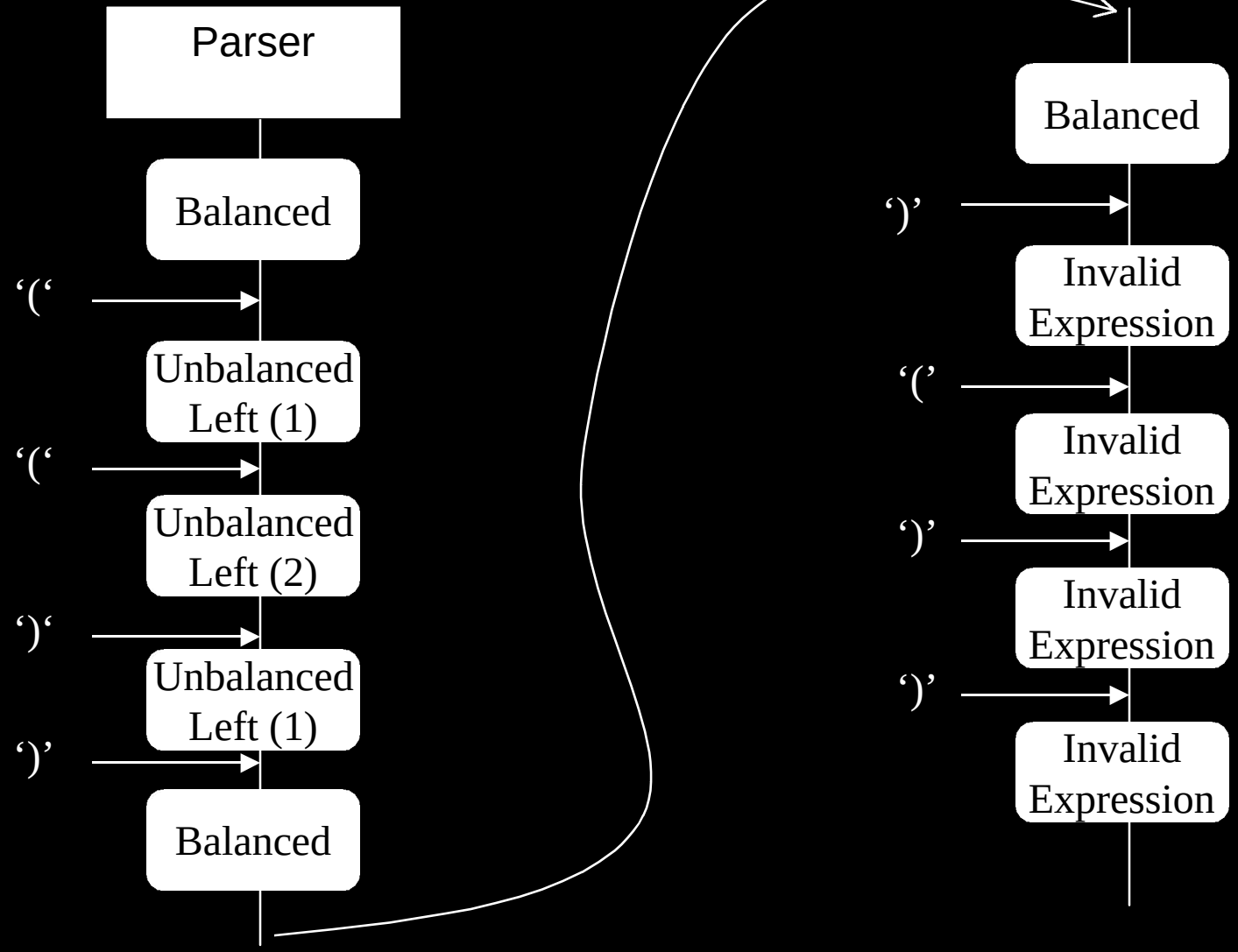
# Example

- Balance ‘( ( () () ) )’



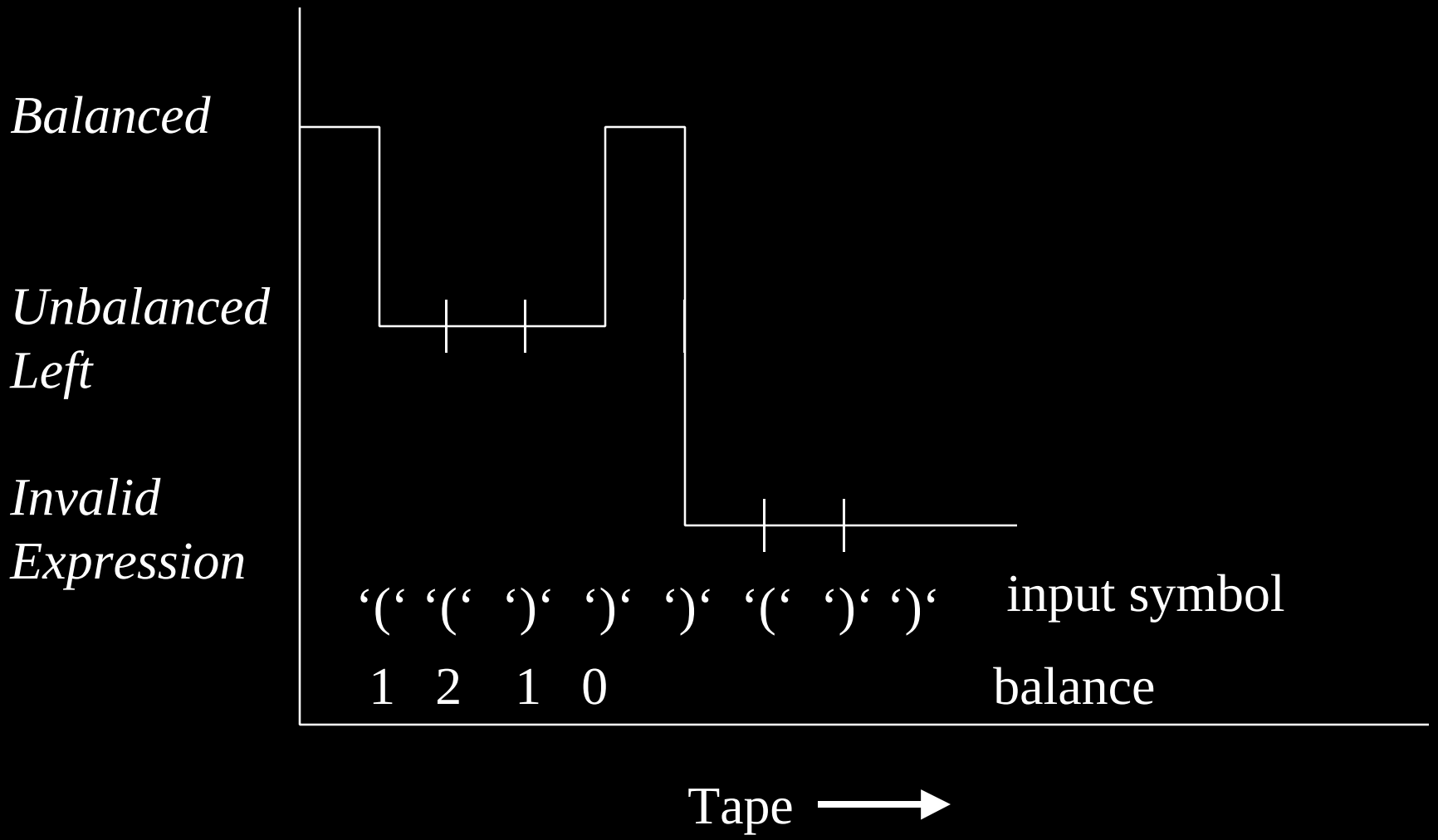
# Example

- Parse ‘( ( ) ) ) ( ) ) ’



# Example

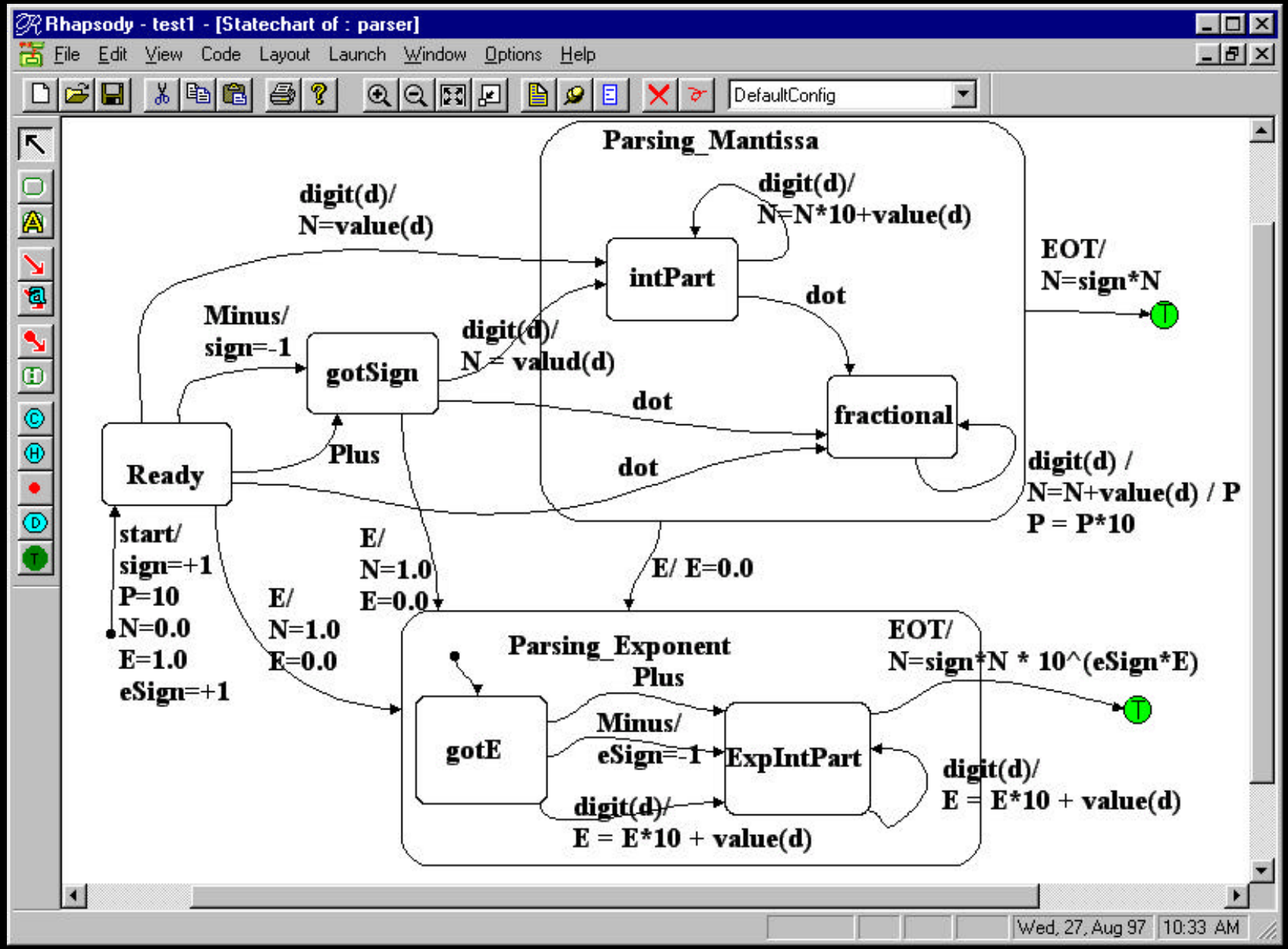
- Balance ‘( ( ) ) ) ( ) ) ’



# Parsing Real Numbers

- Example real numbers
  - 1
  - -1
  - 2.3
  - 1E+17
  - -1E-5
  - 1.4323E3
  - +E-3
- This is a regular grammar (similar to a right-linear grammar)
- Can we build a FSM to produce a binary equivalent of a string representing a real number?

# Real Number Parser

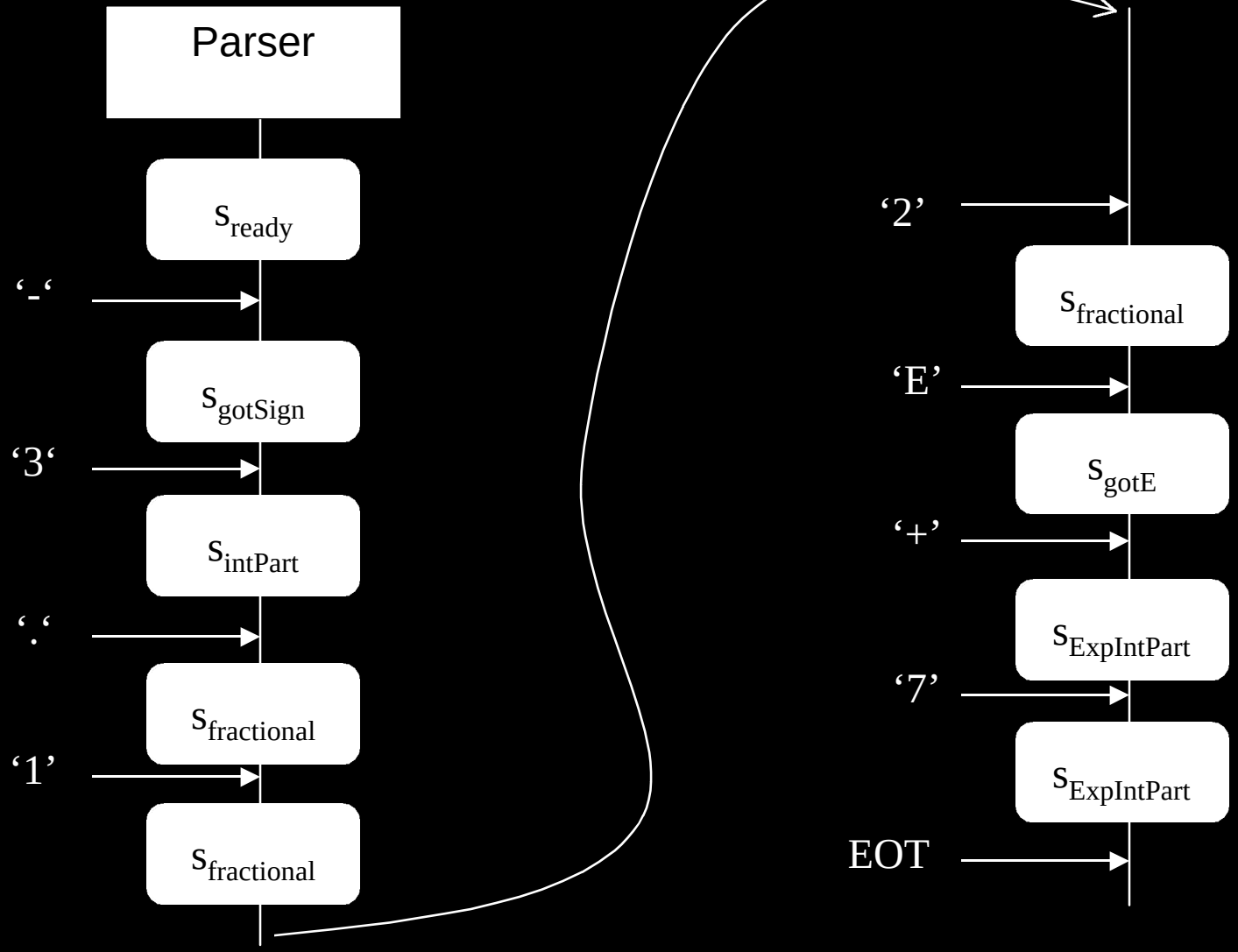


# Real Number Parser

| | digit(d) | '+' | '-' | '.' | 'E' |
|---|---|---|---|---|---|
| **$S_{ready}$** | $S_{intPart}$ | $S_{gotSign}$ | $S_{gotSign}$ | $S_{fractional}$ | $S_{gotE}$ |
| **$S_{gotSign}$** | $S_{intPart}$ | - | - | $S_{fractional}$ | $S_{gotE}$ |
| **$S_{intPart}$** | $S_{intPart}$ | - | - | $S_{fractional}$ | $S_{gotE}$ |
| **$S_{fractional}$** | $S_{fractional}$ | - | - | - | $S_{gotE}$ |
| **$S_{gotE}$** | $S_{expIntPart}$ | $S_{expIntPart}$ | $S_{expIntPart}$ | - | - |
| **$S_{expIntPart}$** | $S_{expIntPart}$ | - | - | - | - |

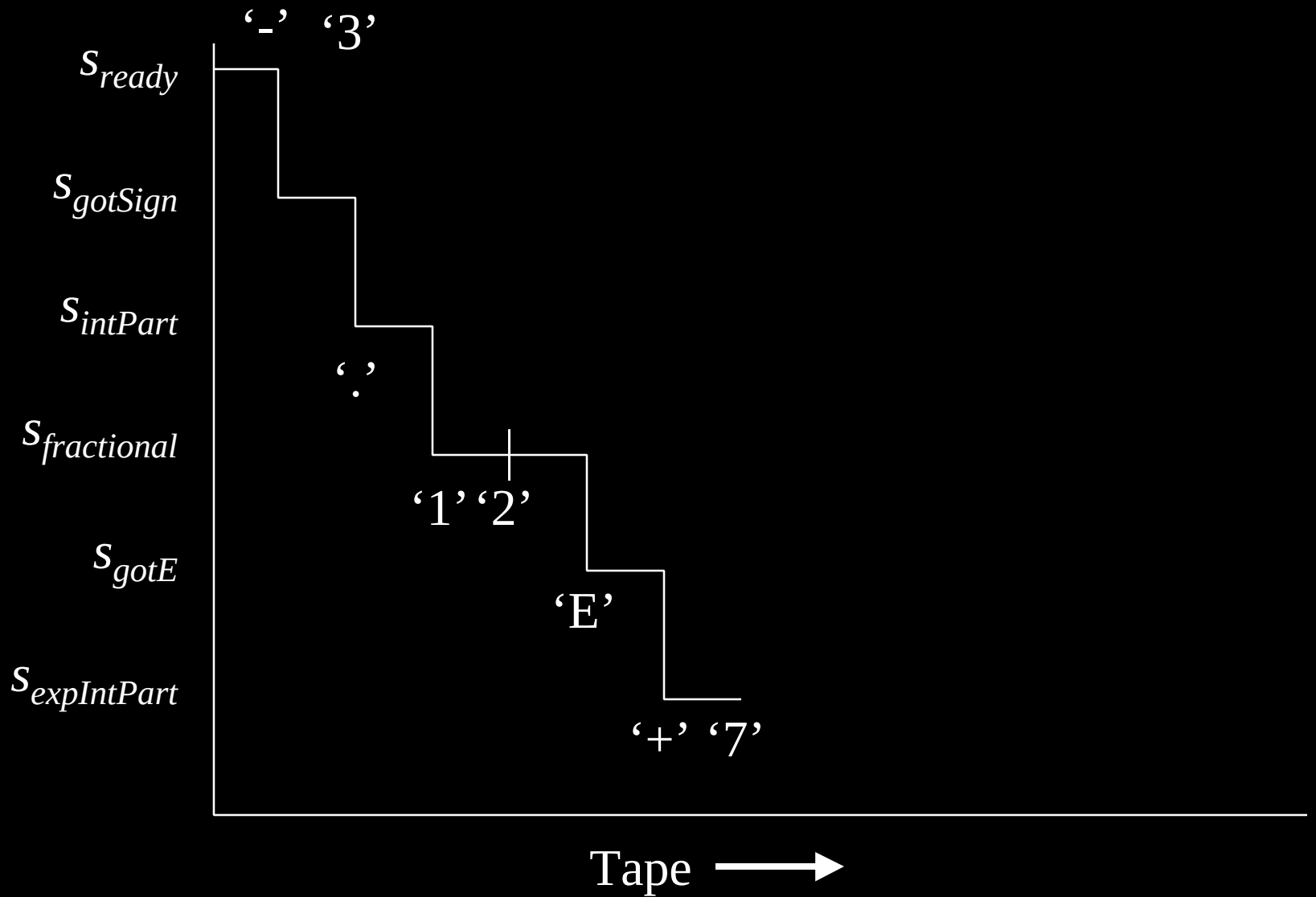
# Real Number Parser

- Parse '-3.12E+7'



# Real Number Parser

- Parse ‘-3.12E+7’



# 10-Key Calculator

- Building a FSM to evaluate arithmetic expressions is a much harder problem than balancing parentheses
  - How should one evaluate:
    - 1 * 2 + 3
    - 1 + 2 * 3
    - 1 - -3
    - 1 * (2 + 3)

# Calculator Grammar

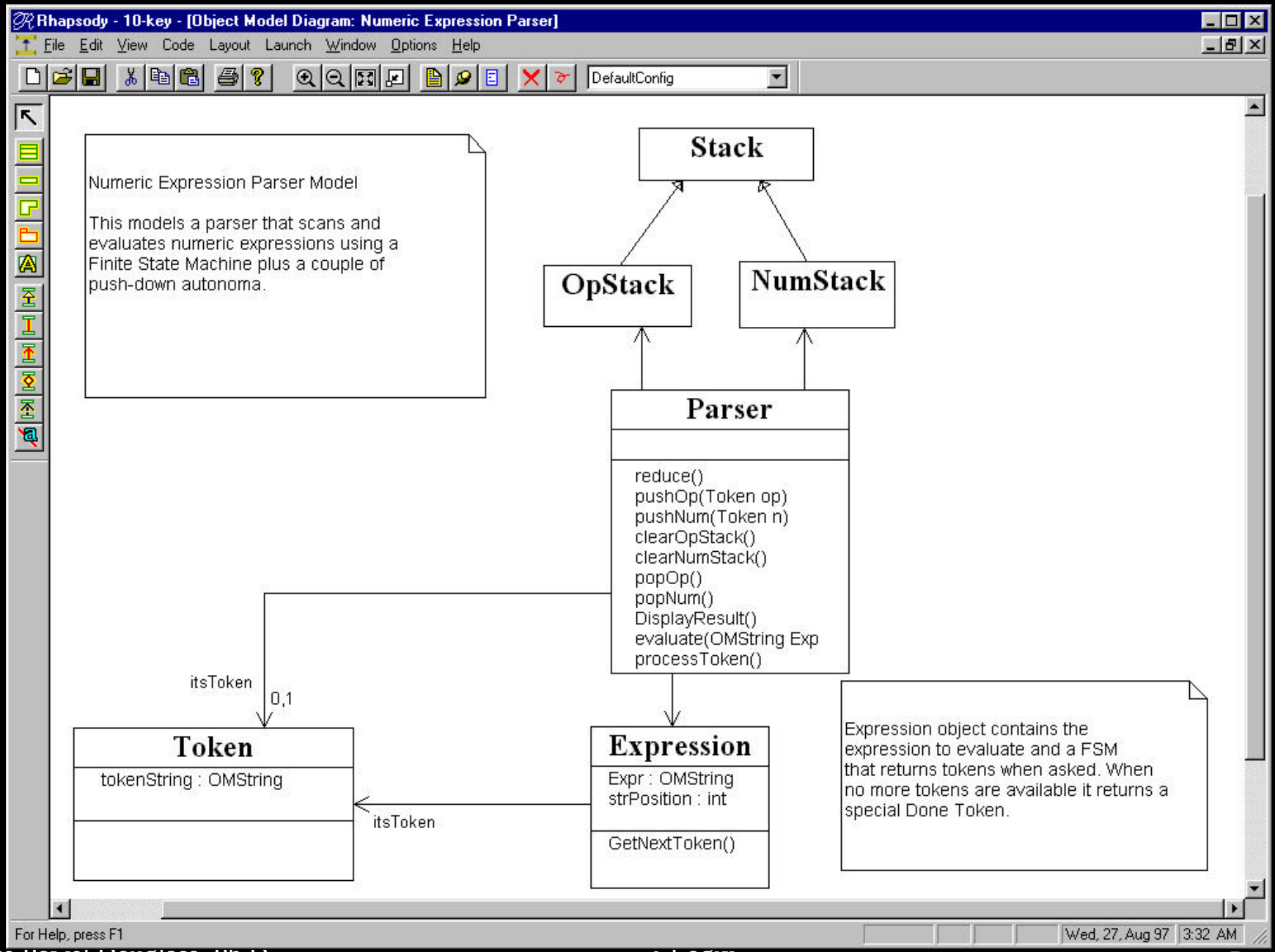
This is a context-free grammar:

| | | |
|---|---|---|
| Expr | → Expr + Term | #PLUS |
| | → Expr - Term | #MINUS |
| | → Term | |
| Term | → Term * Fact | #MULTIPY |
| | → Term / Fact | #DIVIDE |
| | → Fact | |
| Fact | → Primary | |
| Primary | → ( Expr ) | #PARENTHESES |

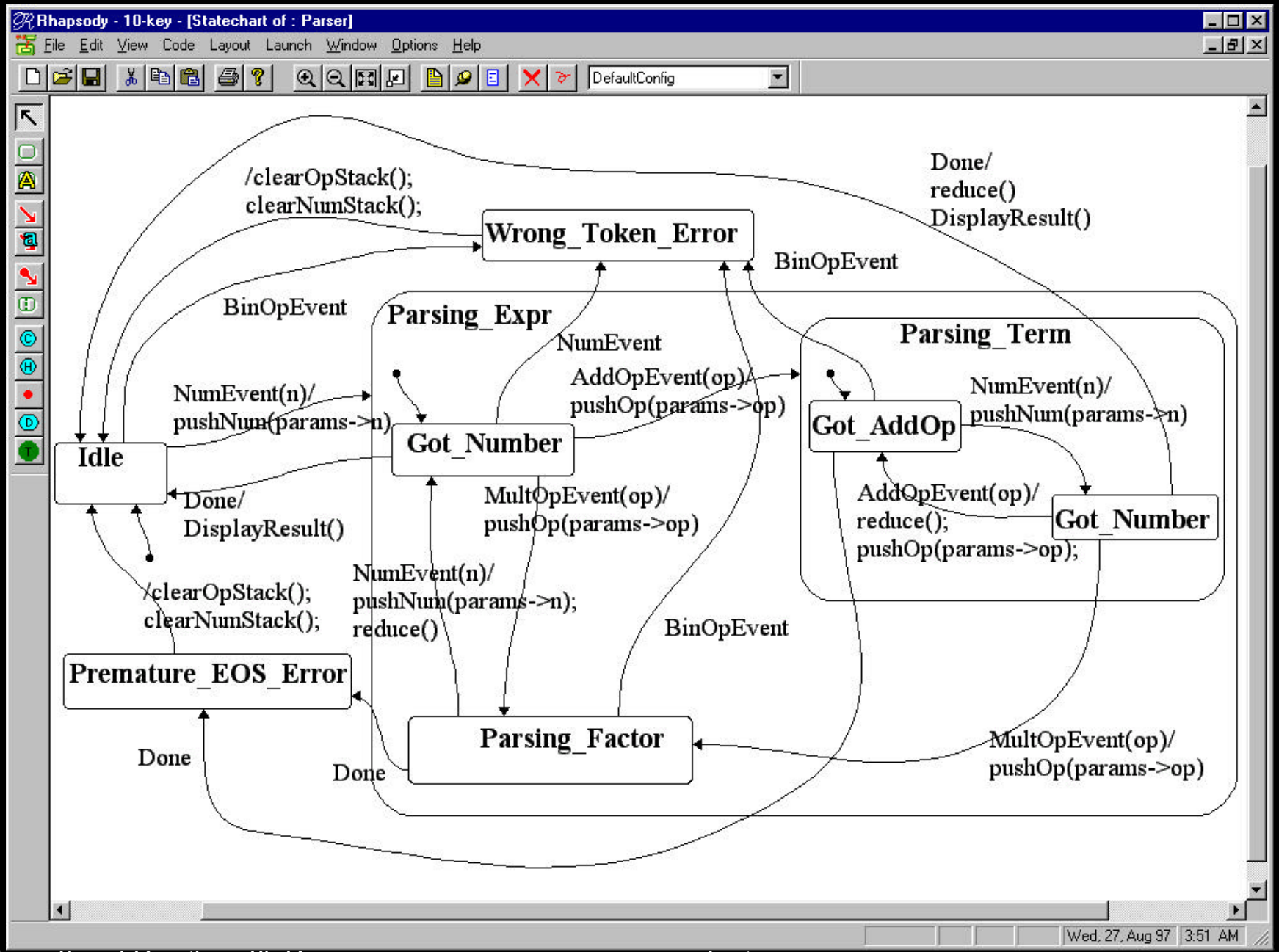
# Calculator Solution

- We will use an object design here with 2 stacks
  - operator stack
  - number stack

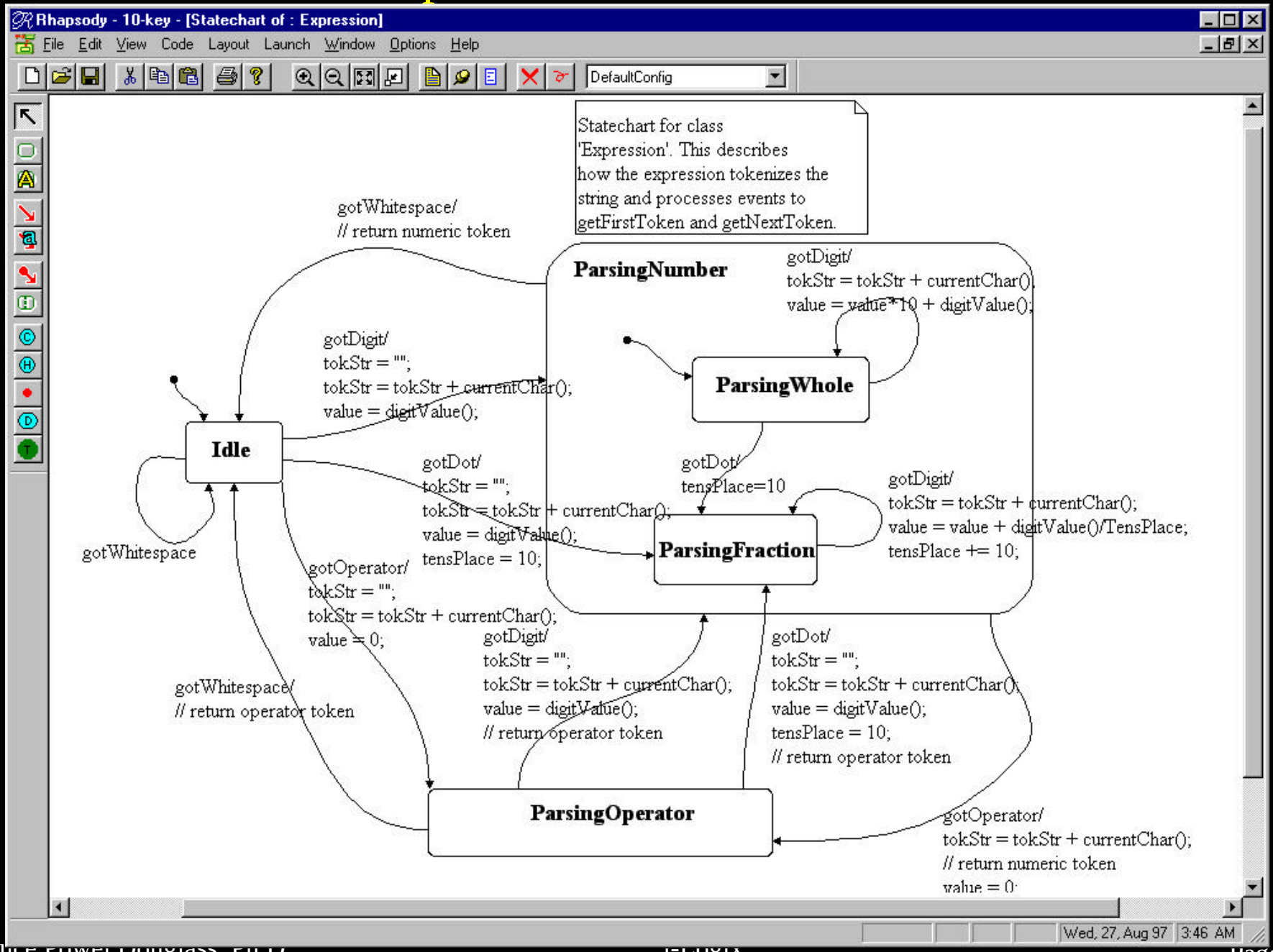
# Calculator Object Model



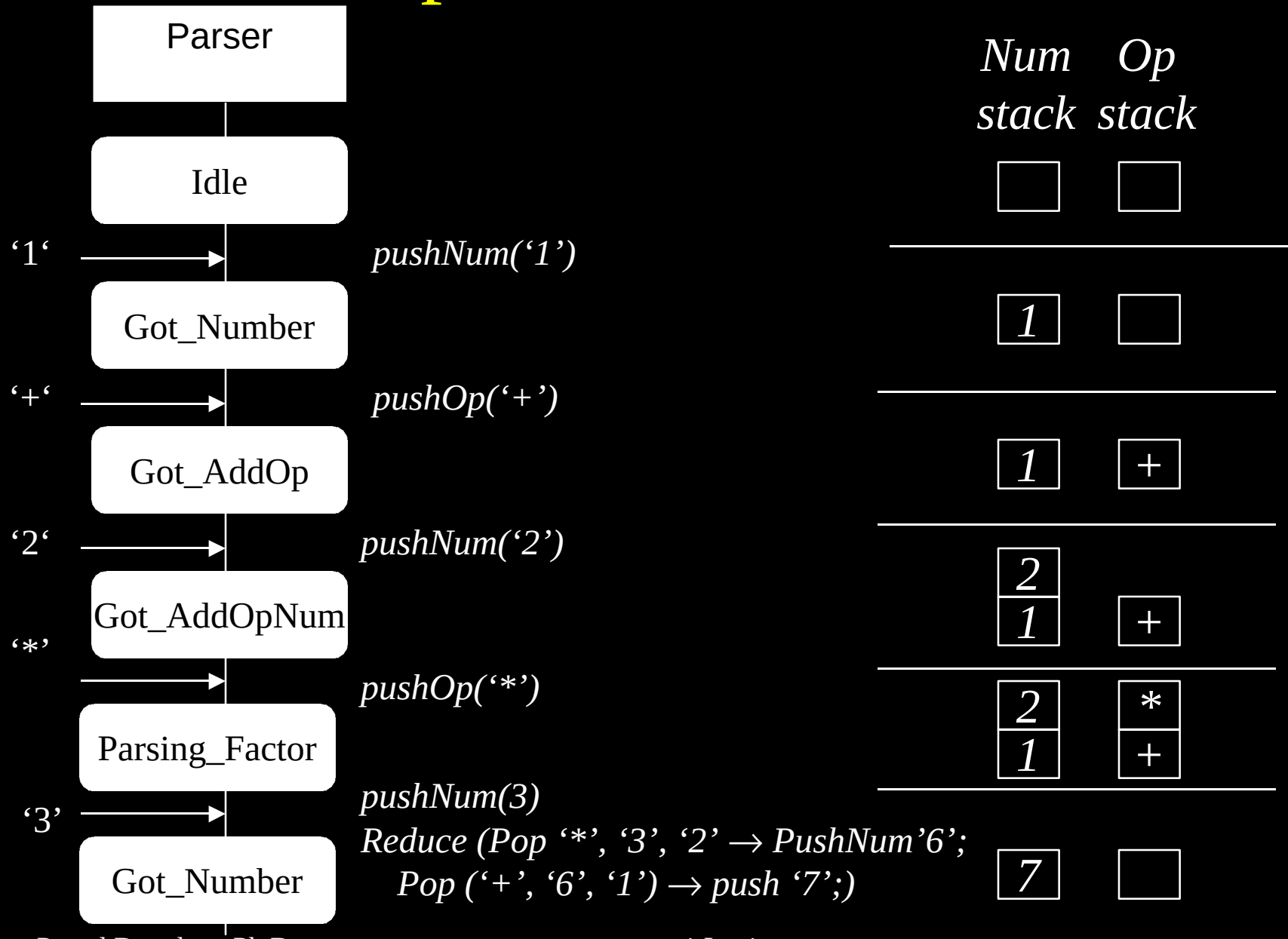
# Parser State Model



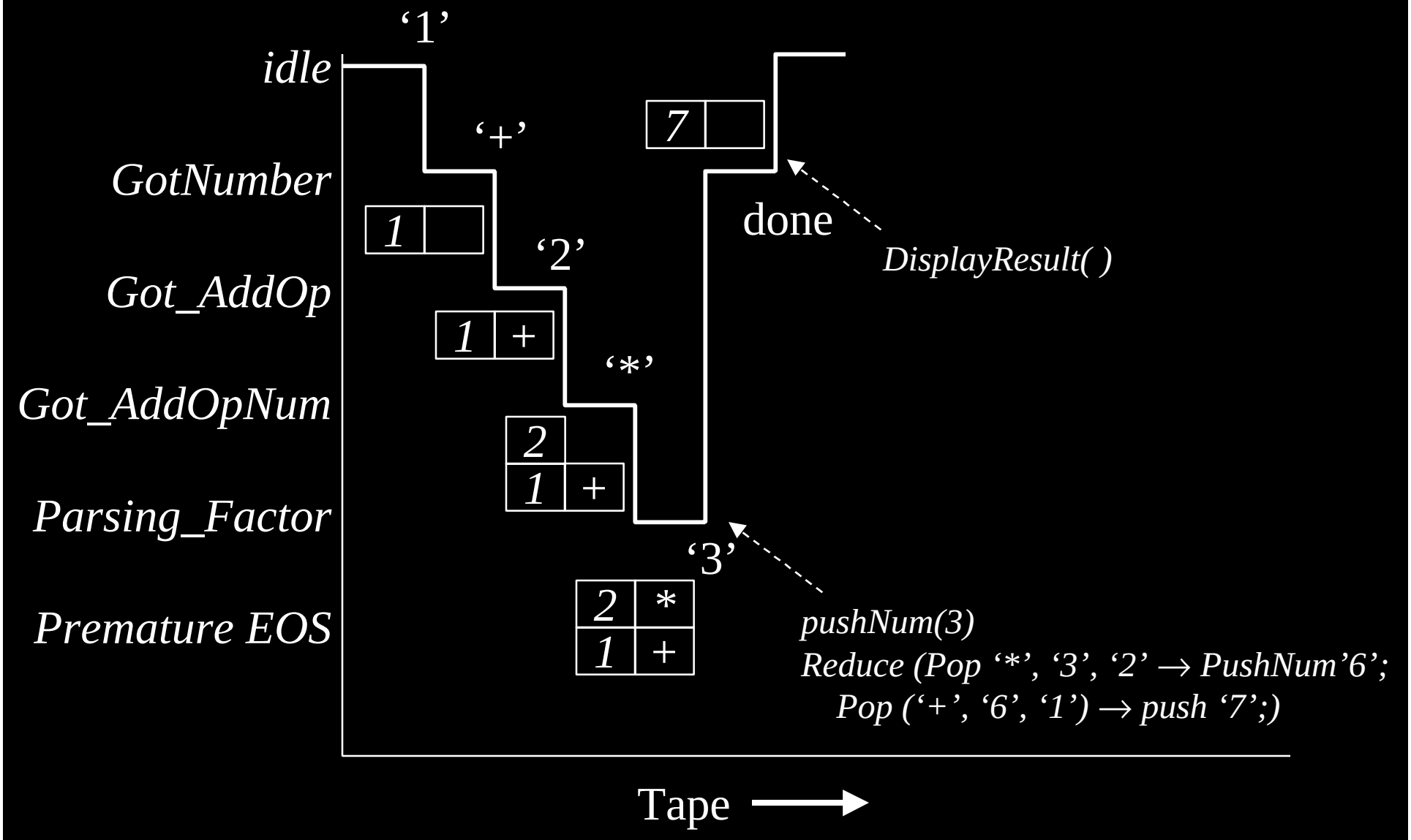
# Expression State Model



# Example: Parse “1+2*3”



# Example: Parse “1+2*3”



# Problems with Model

- How to handle Parenthesis?
  - Note that when we expect an operator (+ - * /) we can get a Right Parenthesis, as in

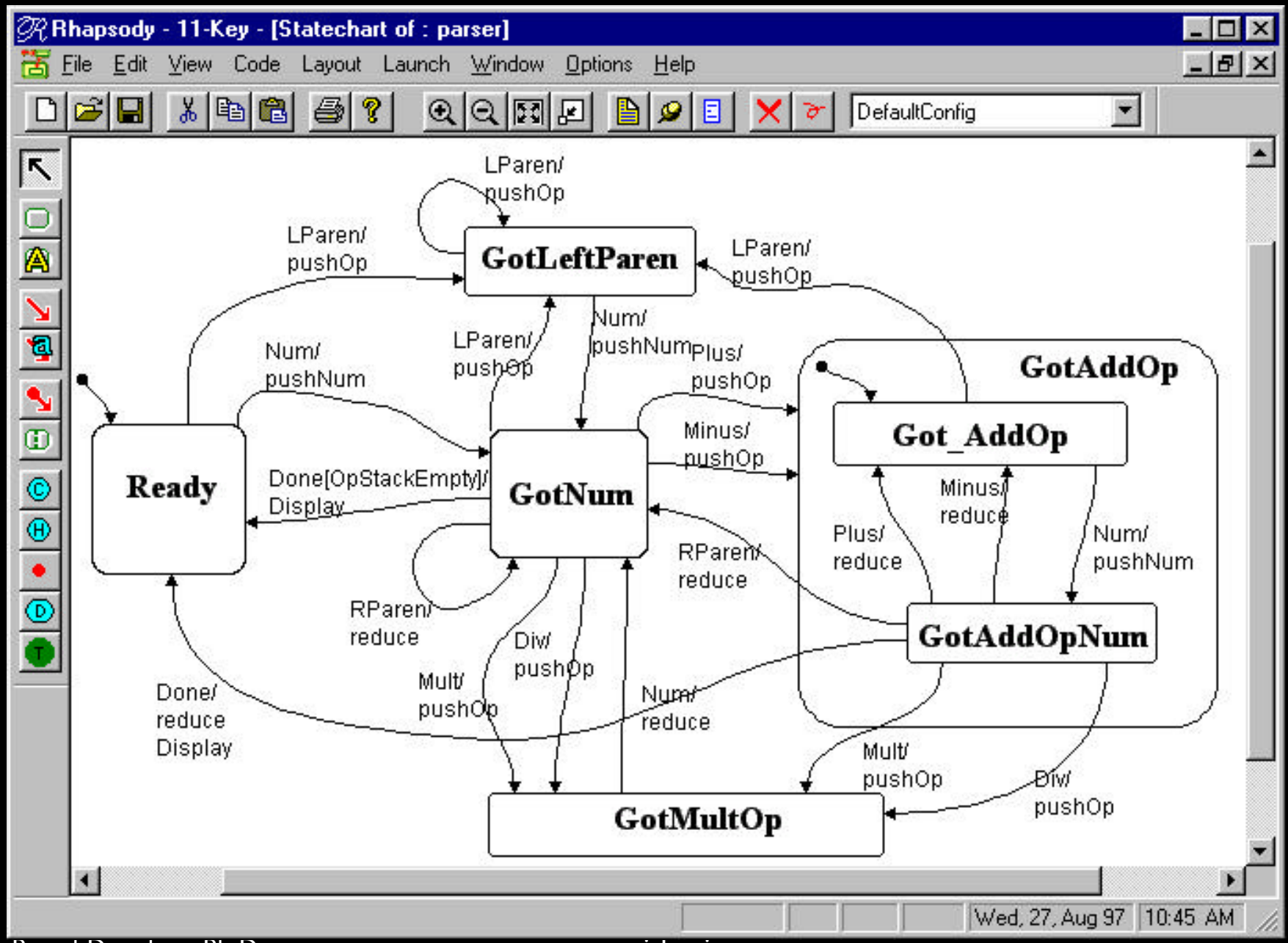
    (1+2)  (3+4-6)

    ↑  ↑
  - Note that when we expect a number we can get a Left Parenthesis, as in

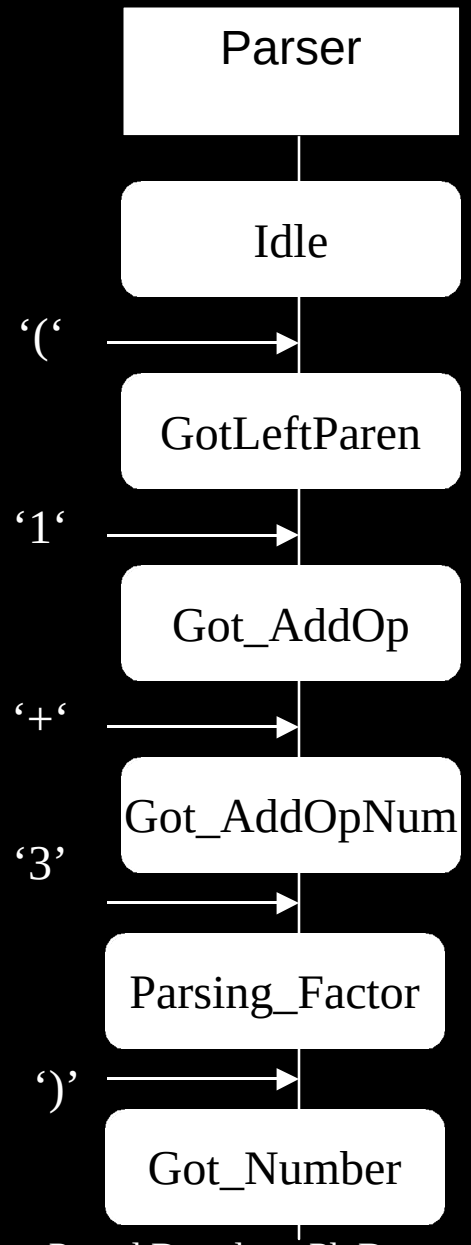
    (2)  (1+3*(3+4))

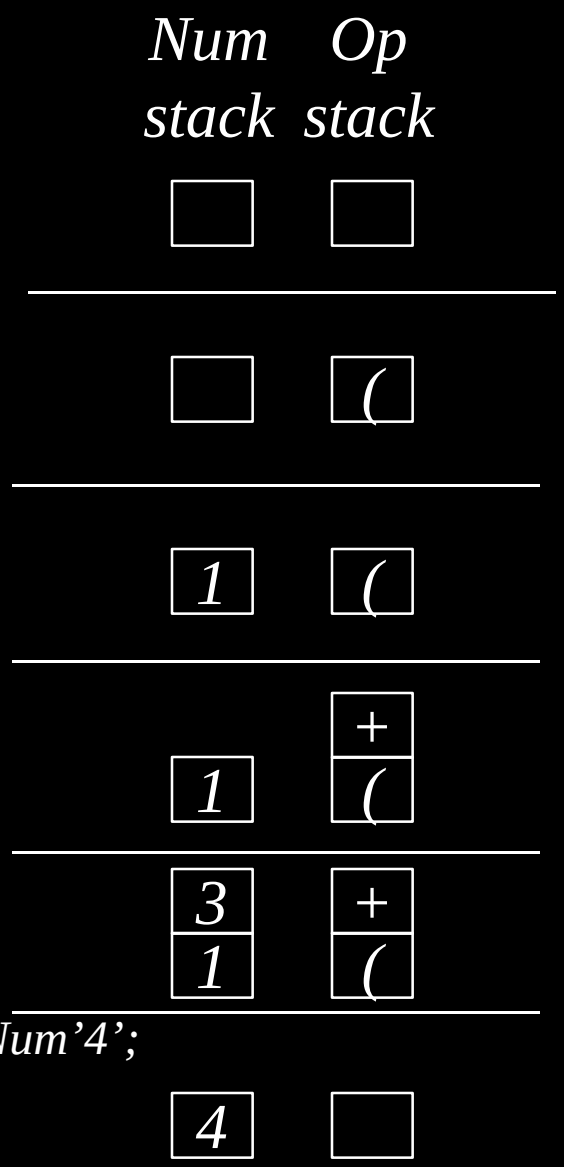
    ↑  ↑  ↑
  - An operator or Right Parenthesis can follow a Right Parenthessis but not a number  ")17" ← illegal
  - A number or Left Parenthesis can follow a Left Parenthesis but not an operator  "(*" ← illegal

# Improved Parser State Model

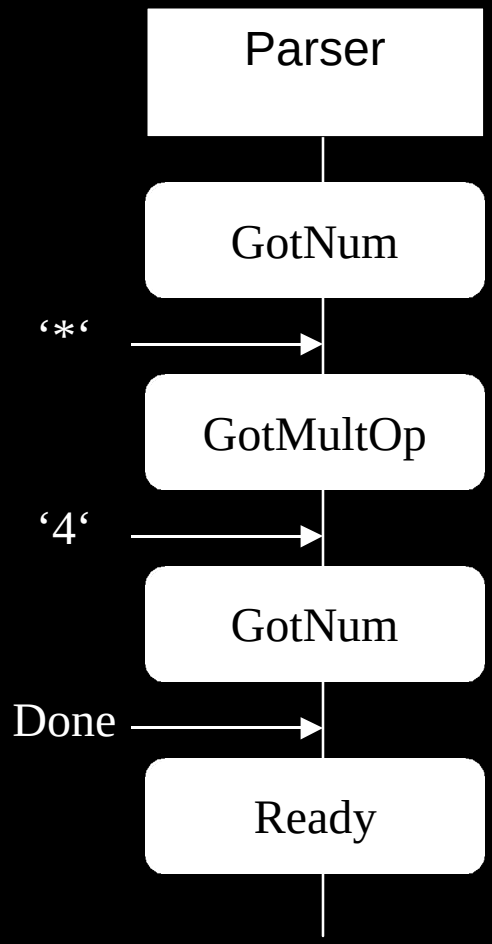


# Example: Parse "(1+3)*4"



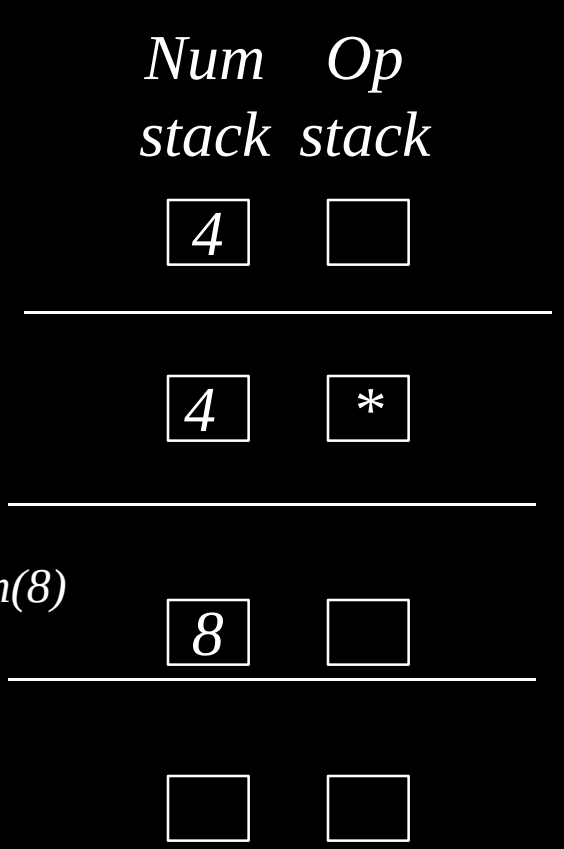
# Example: Parse "(1+3)*4" (cont)




*pushOp('*')*

*pushNum('4')*
*Reduce(Pop '*', '4', '4'; pushNum(8)*

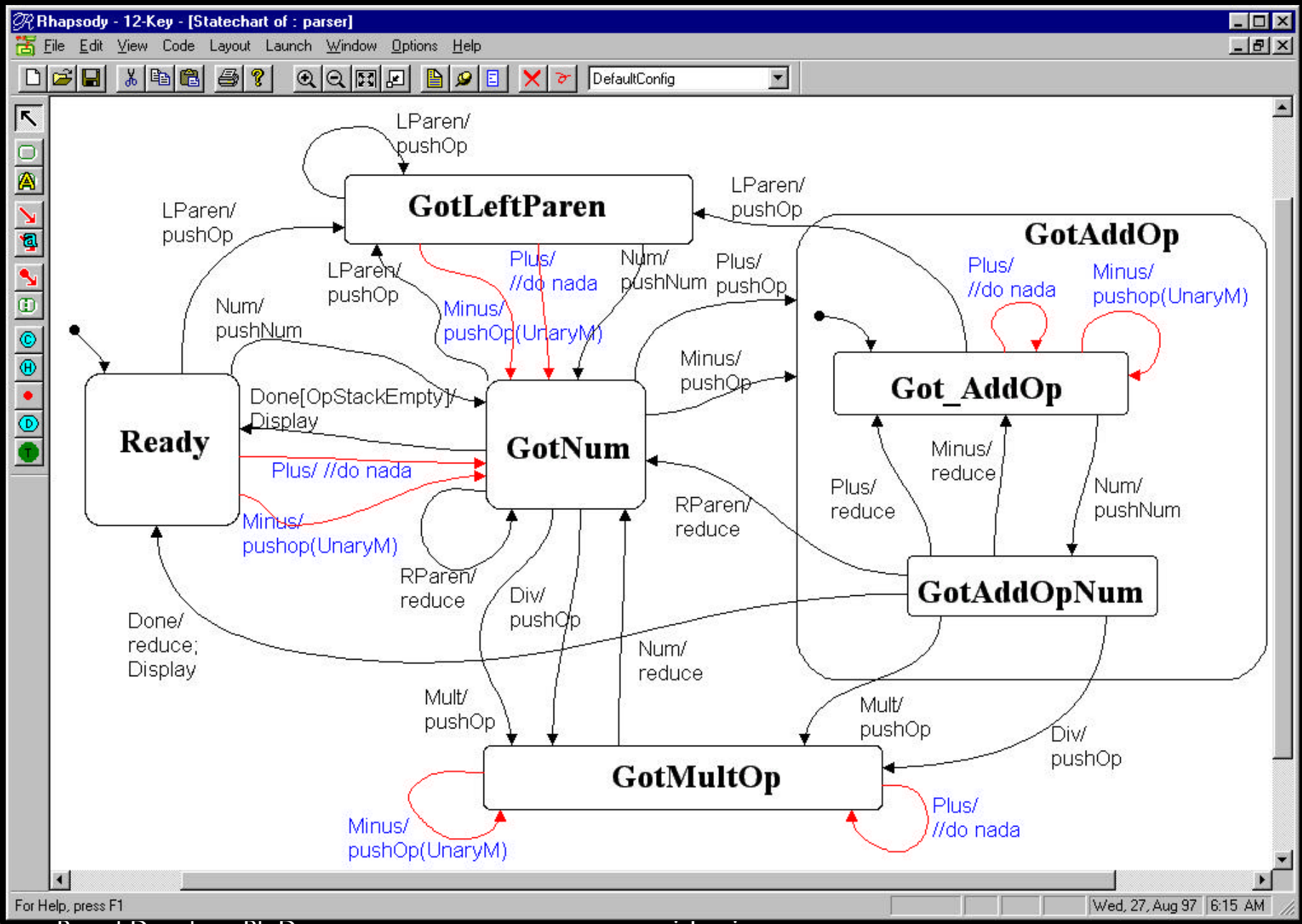
*Display(d = PopNum; show(d); )*



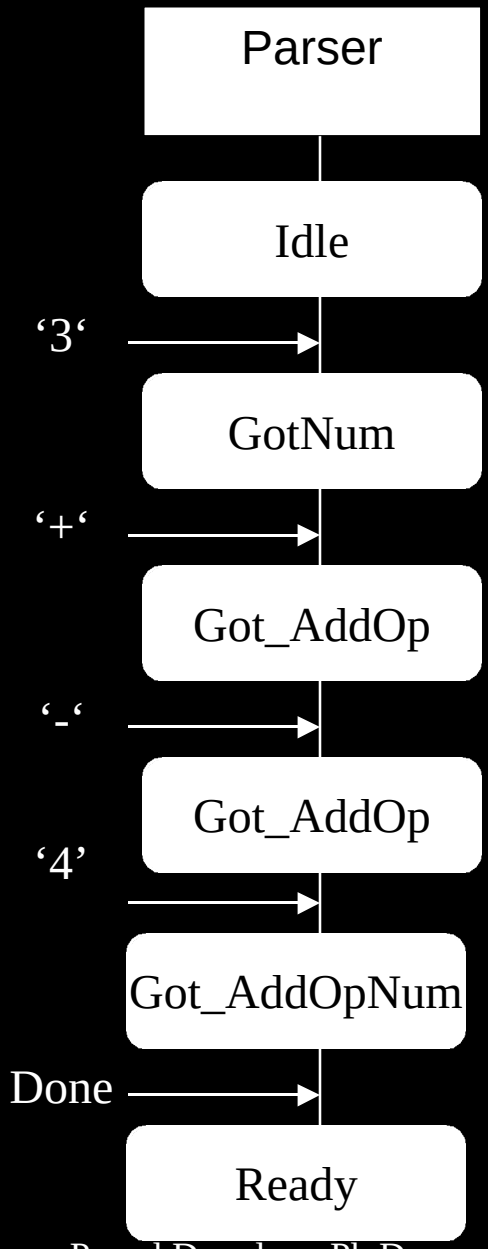
# Problems with Model (2)

- How do we handle unary operators?
  - -17
  - +25
  - 4 - -6
  - -4 * +19
- Note that if an addOp {‘+’, ‘-’} occurs when expecting a number, it should be treated as a unary operator (i.e. sign)

# Improved Parser State Model (2)



# Example: Parse "3+-4"



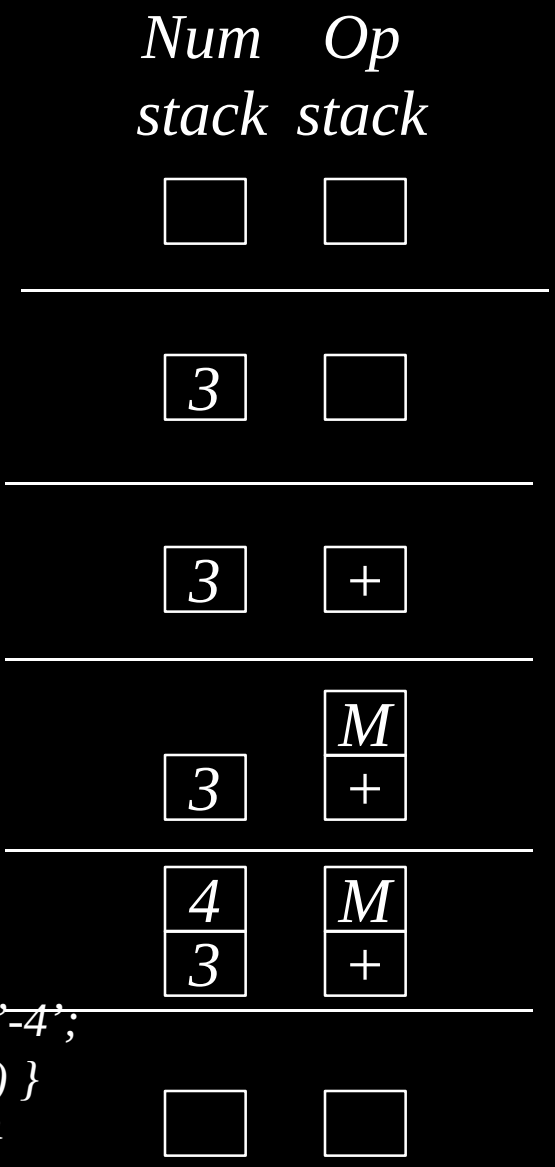

*pushNum('3')*

*pushOp('+')*

*pushOp(UnaryM)*

*pushNum('4')*

*Reduce {Pop 'M', '4' → PushNum'-4';*
*Pop ('+', '-4', '3') → push '-1';) }*
*Display { d =Popnum(); show(d); }*



# Exercise for the Reader

- Validate the state machine for following expressions
  - (2+3)/(7-9)
  - 2*( ( (3-2) * (4+1) ) - 5)
- Add error states and transitions to identify the following errors
  - )3-2
  - 1 - *3
  - 4+5)-7

# Object-Oriented State Patterns

# Simple Approach #1

- Use nested CASE statements to implement state

```
case (state1) {
        switch(event) {
                case e1: ….; break;
                case e2: ….; break;
                }
case state2 :
        switch(event) {….
```

- Problems
  - Performance
  - Scalability

# Simple Approach #2

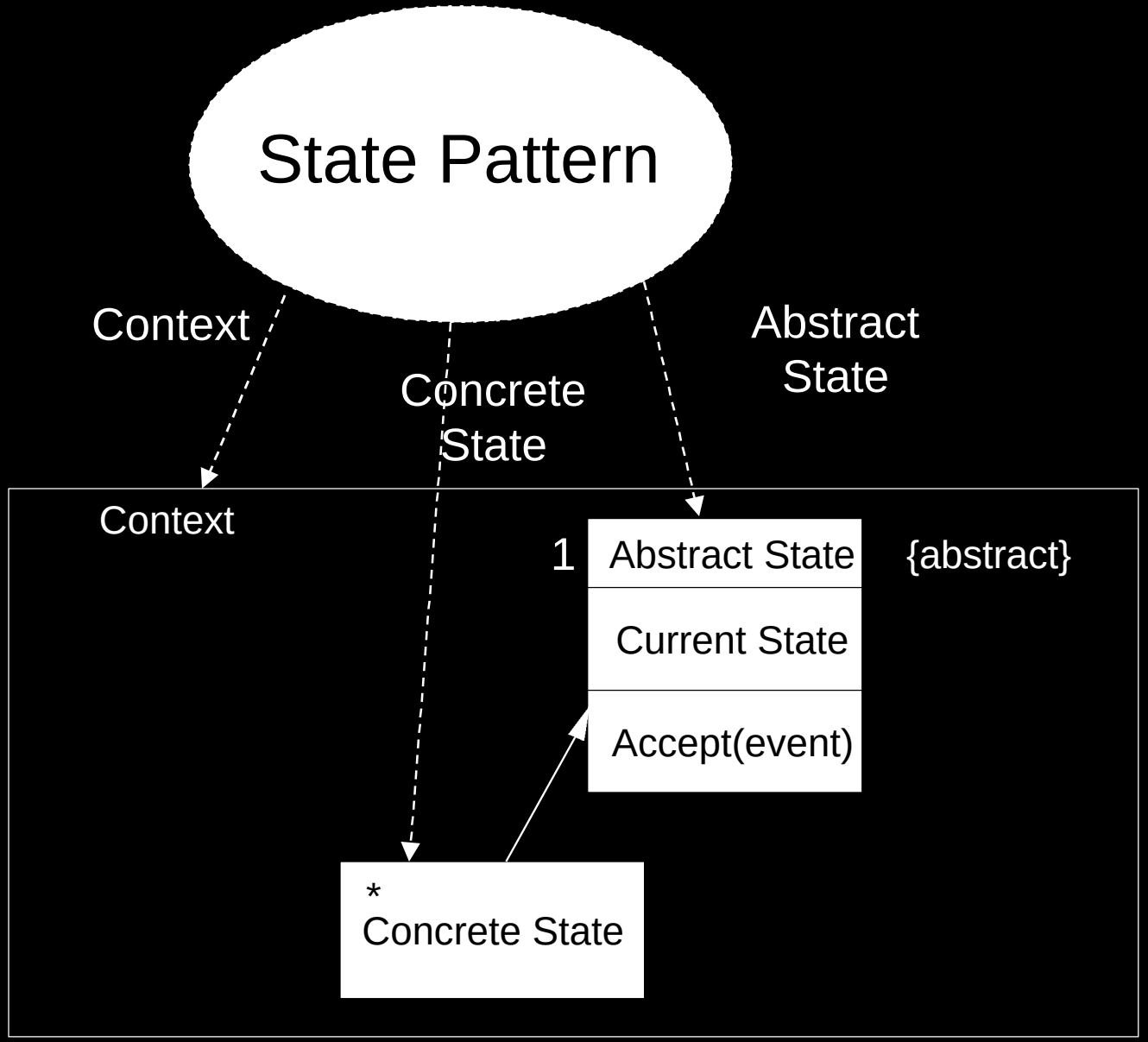
- Use a single state variable (class attribute) which holds the current state
- Event acceptor operations
  - actions of event acceptor operations vary depending on the value of the state variable
  - event acceptor operations update this state variable to change state

# Simple Approach #2

- Example

```
myClass::AcceptTurn(int nClicks) {
        if (stateVar = OPERATIONAL) {
                display(nClicks);
                clicks += nClicks;
                if (clicks > MAX)
                        stateVar = OVERFLOW;
        }; // end if
};
```

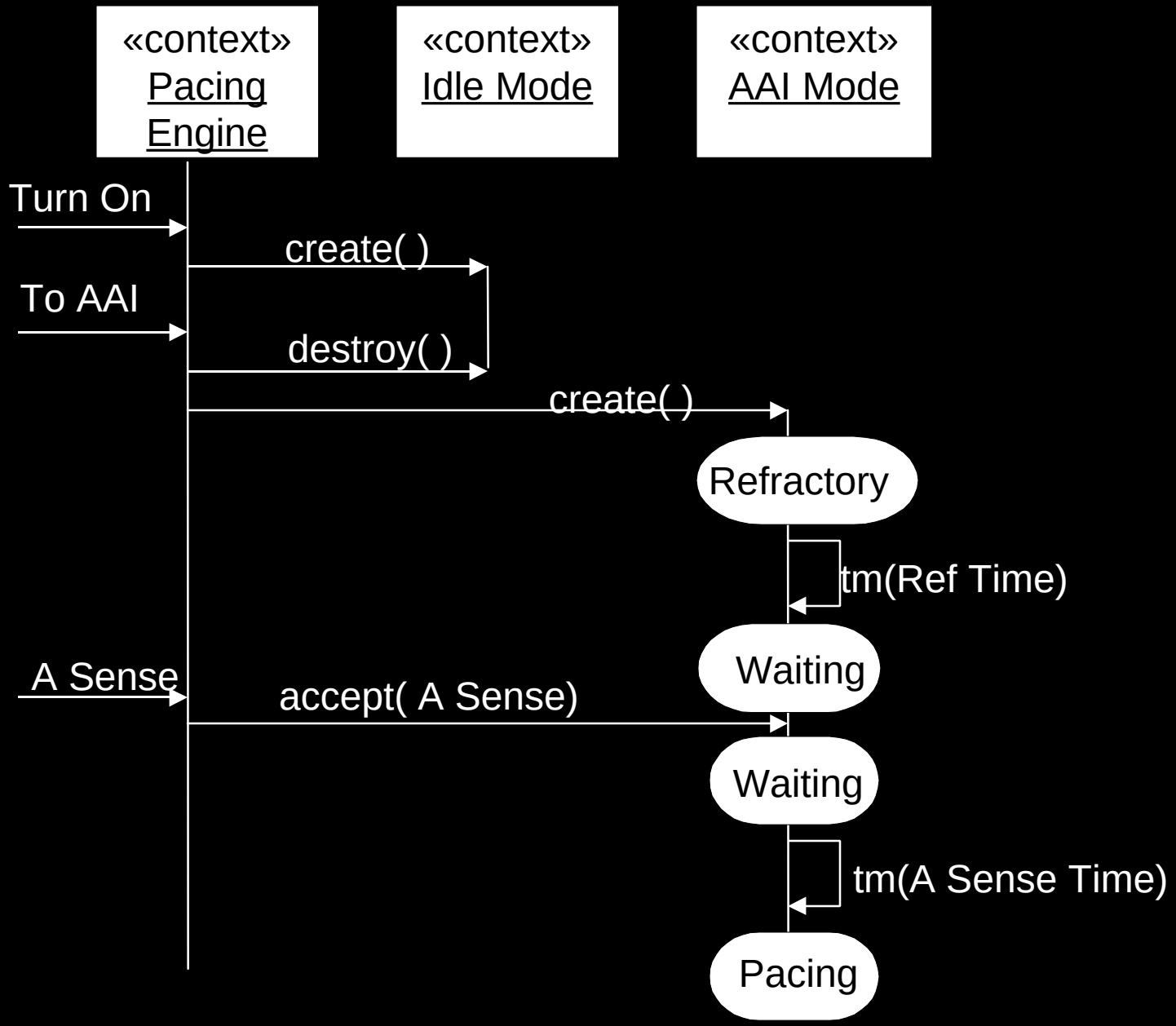
# State Pattern



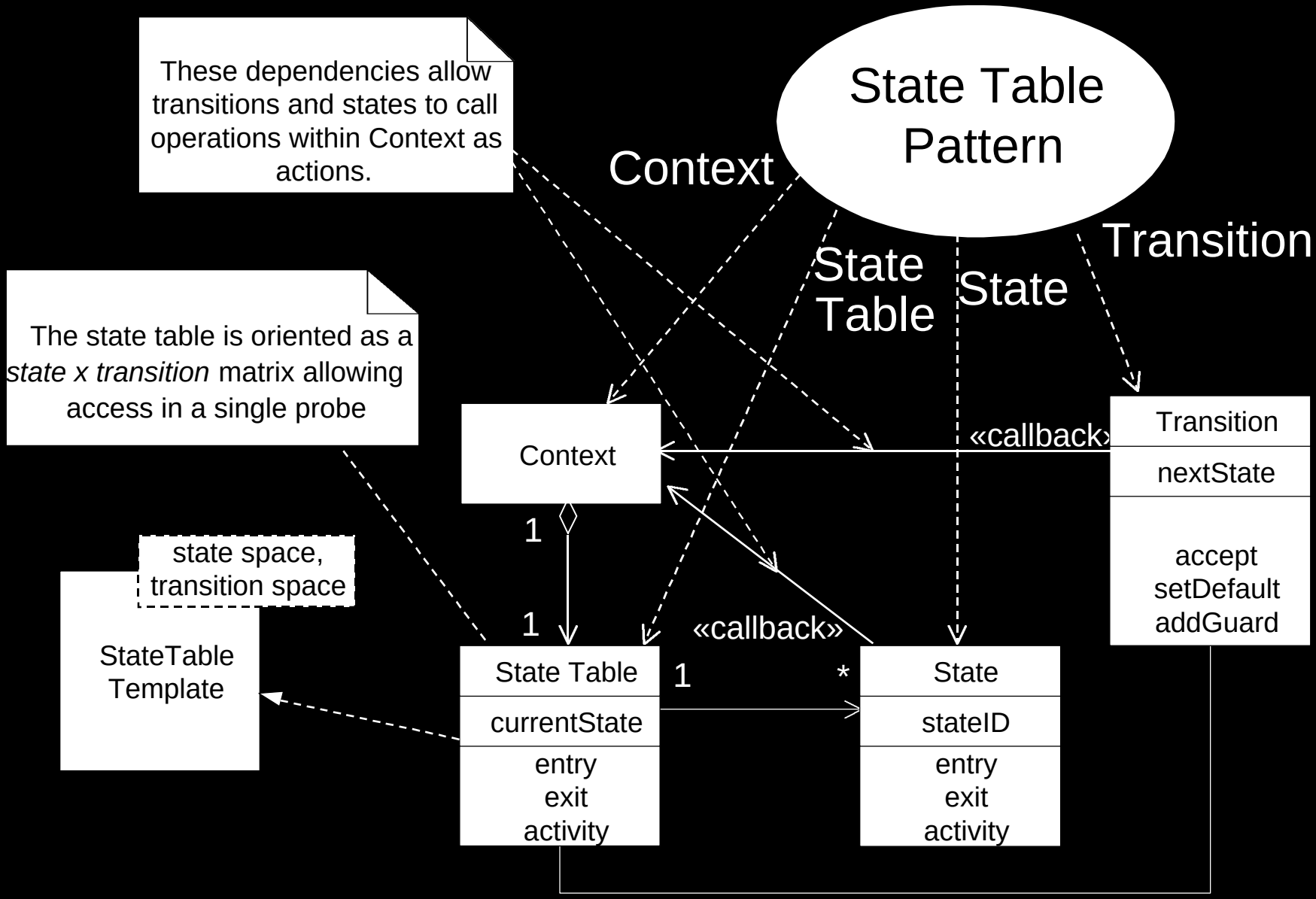
# State Pattern

- Good when some set of states are long lasting
- Good to optimize rapidly-changing states by making less-frequent state changes more expensive

# State Pattern



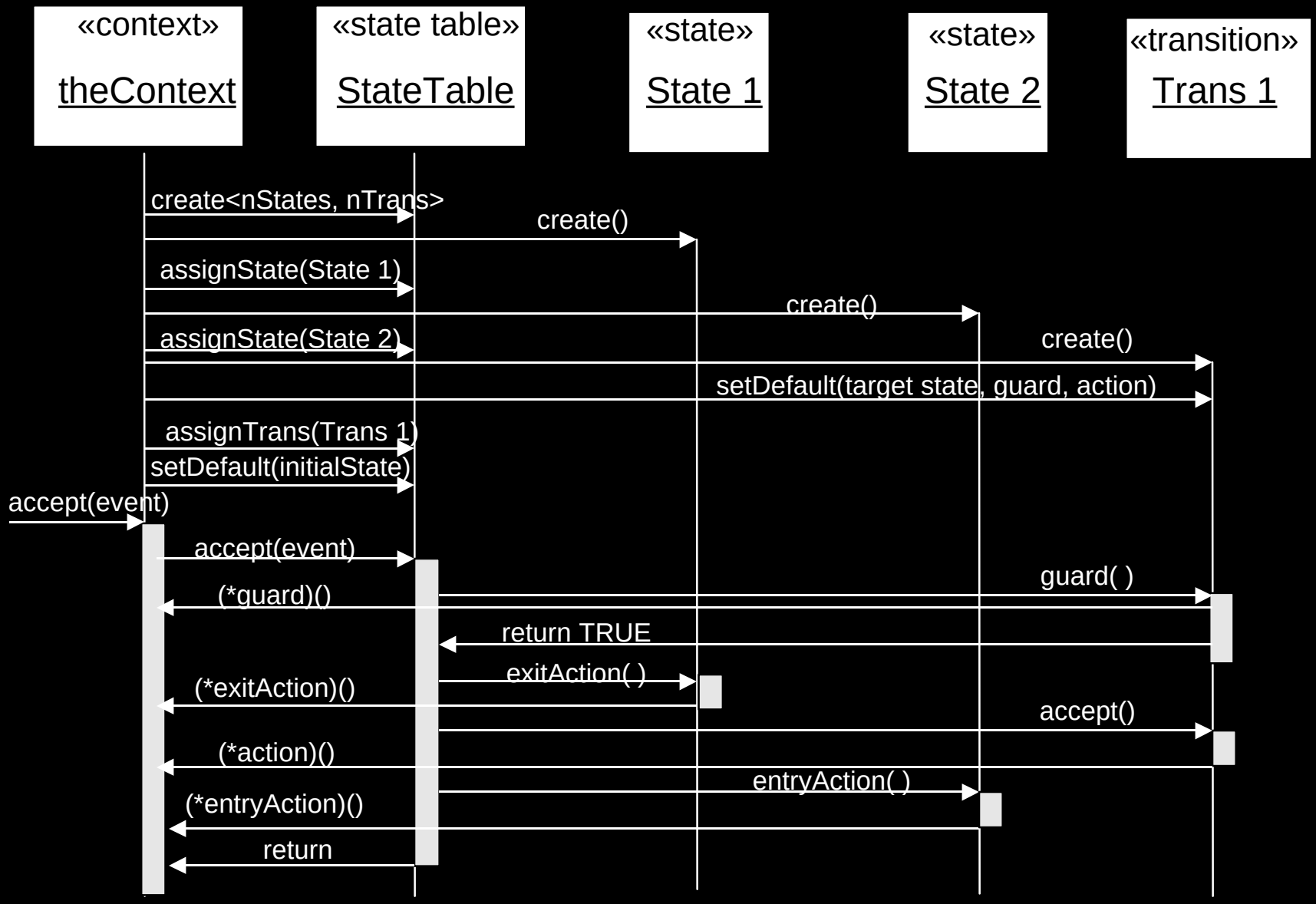
# State Table Pattern



# State Table Pattern

- Good for large state spaces
- Good for constant run-time performance
- More complex and expensive to set up

# State Table Pattern



# References of Interest


- Douglass, Bruce Powel *Real-Time UML: Efficient Objects for Embedded Systems*. Reading, MA: Addison-Wesley-Longman, Dec. 1997
- Douglass, Bruce Powel *Doing Hard Time: Using Object Oriented Programming and Software Patterns in Real Time Applications.* Reading, MA: Addison-Wesley-Longman, Spring 1999
- Brookshear, J. Glenn *Theory of Computation: Formal Languages, Automata, and Complexity*. Redwood City, CA: Benjamin/Cummings, 1989
- Levy, Leon S. *Fundamental Concepts of Computer Science: Mathematical Foundations of Programming*. New York: Dorset Publishing, 1988
- Harel, David *Algorithmics: The Spirit of Computing*. Reading, MA: Addison-Wesley-Longman, 1993

# Summary

- Objects have behavior
  - Simple
  - Continuous
  - State-driven
- Modeling objects as Finite State Machines simplifies the behavior
- States apply to objects
- FSM Objects spend all their time in exactly 1 state (discounting orthogonal substates)

# Summary

- States are *disjoint ontological conditions that persist for a significant period of time.*
- States are defined by one of the following:
  - The values of all attributes of the object
  - The values of specific attributes of the object
  - Disjoint behaviors
    - Events accepted
    - Actions performed

# Summary

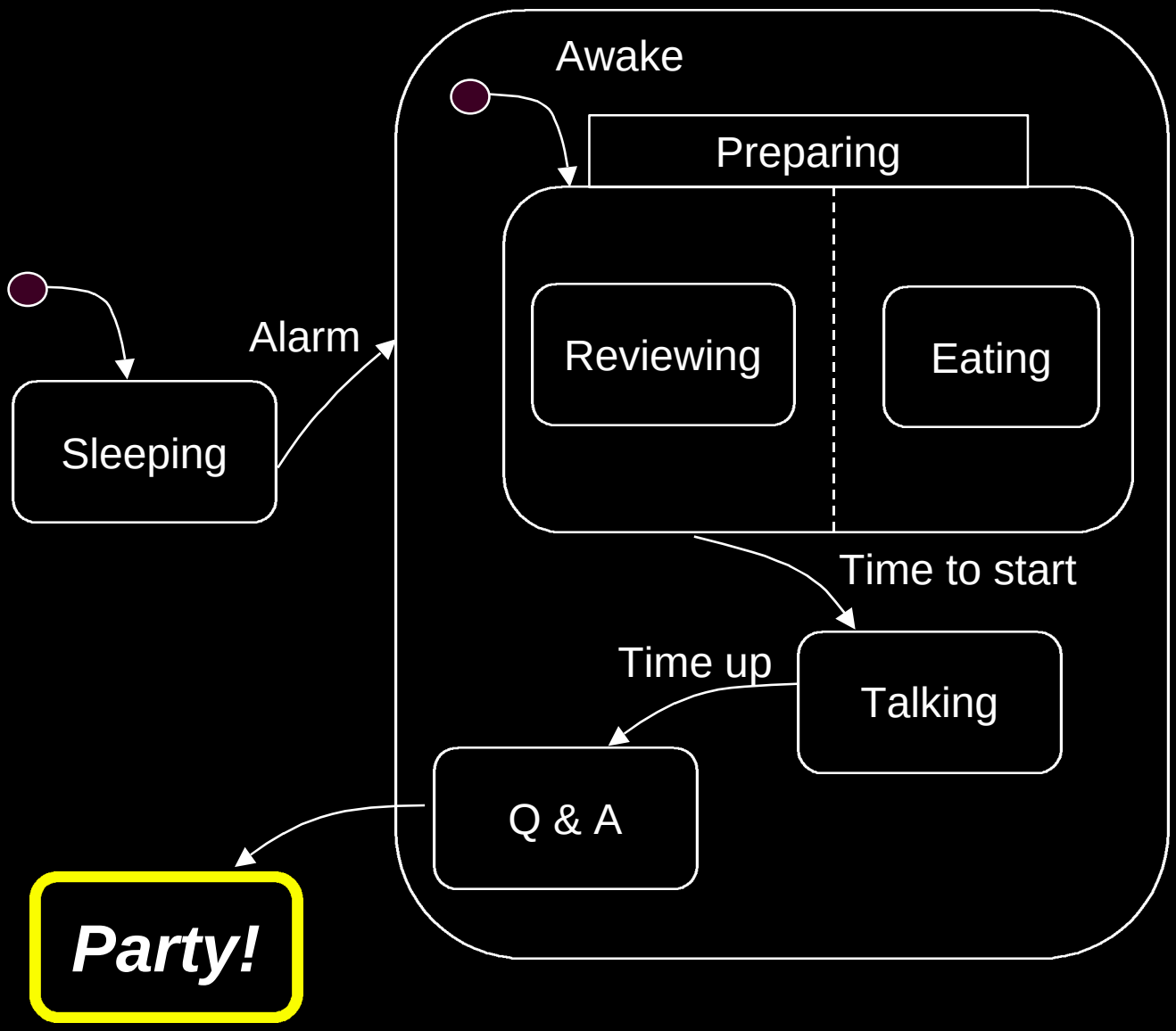
- Transitions are the representation of responses to events within FSMs
- Transitions take an insignificant amount of time
- Actions are functions which may be associated with
  - Transitions
  - State Entry
  - State Exit
- Activities are processing that continues as long as a state is active

# Summary

- Harel statecharts expand standard FSMs
  - Nested states
  - Concurrency
  - Broadcast transitions
  - Orthogonal Components
  - Actions on states or transitions
  - History
  - Guards on transitions

# Summary

- Statecharts show static structural view
  - good “roadmap”
- State tables show static structureal view
  - good for id’ing missing transitions
- Sequence diagrams show scenarios
  - good for looking walking through sequences of transitions
- Timing diagrams show overall timing in scenarios
  - good for looking at timing details of sequences


Awake
Preparing
Reviewing
Eating
Alarm
Sleeping
Time to start
Time up
Talking
Q & A
Party!