# Human Computer Interaction

## Dialog Notations and Design

Lecture#14a

Imran Siddiqi
imran.siddiqi@gmail.com

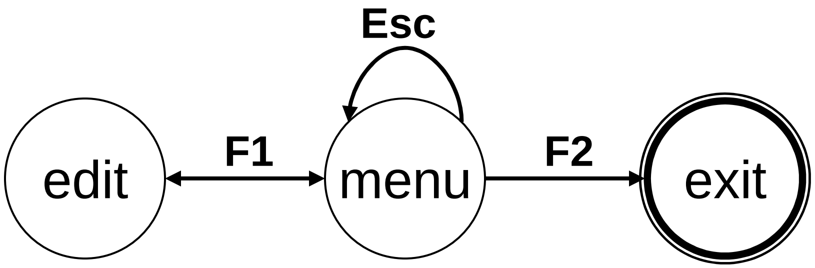
# From Last Time …

- Dialog Notations
  - State Transition Networks
  - Petri Nets
  - State Charts
  - Flow Charts
  - JSD

# Dangerous States

- Word processor: two modes and exit

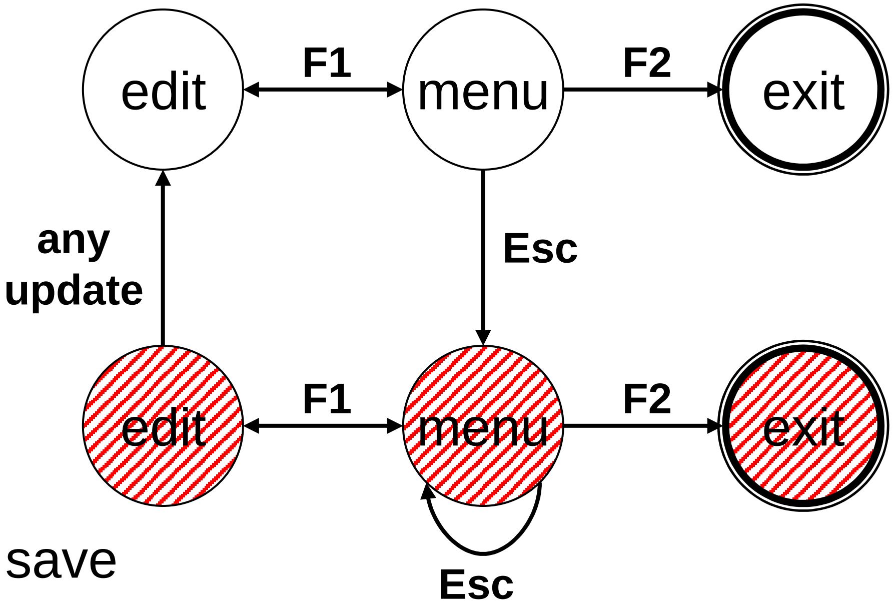
| | | |
|---|---|---|
| F1 | - | changes mode |
| F2 | - | exit (and save) |
| Esc | - | no mode change |



but ... Esc resets autosave

# Dangerous States

- Exit without save ⇒ dangerous state
- Duplicate states - semantic distinction

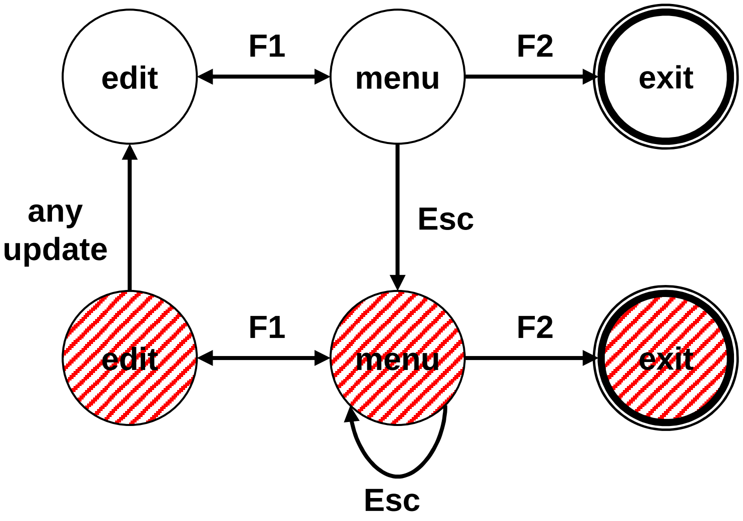


F1-F2 - exit with save

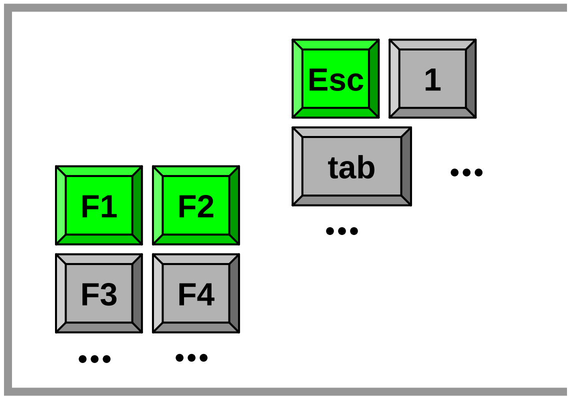
F1-Esc-F2 - exit with no save

# Dangerous States and Key Layout

- Word processor - dangerous states

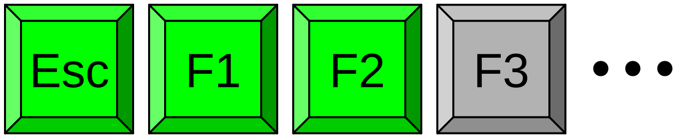


- Old keyboard - OK



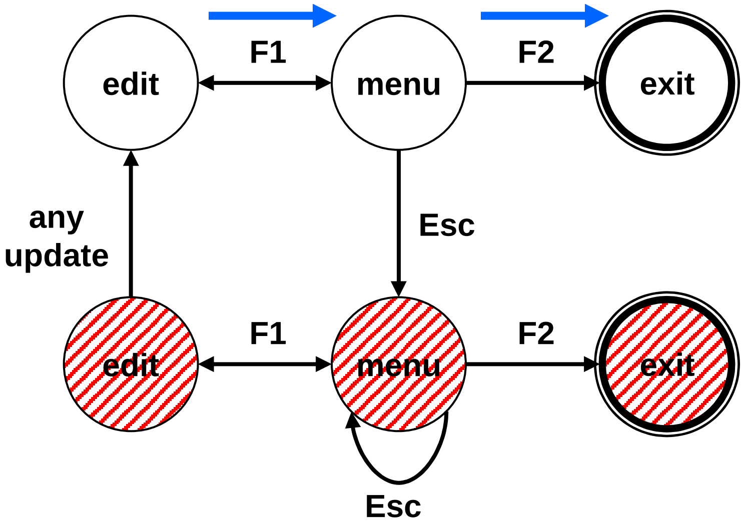
# Dangerous States and Key Layout

- New keyboard layout



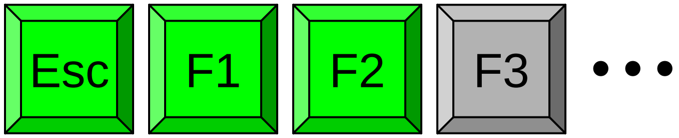
intend F1-F2 (save)

finger catches Esc



# Dangerous States and Key Layout

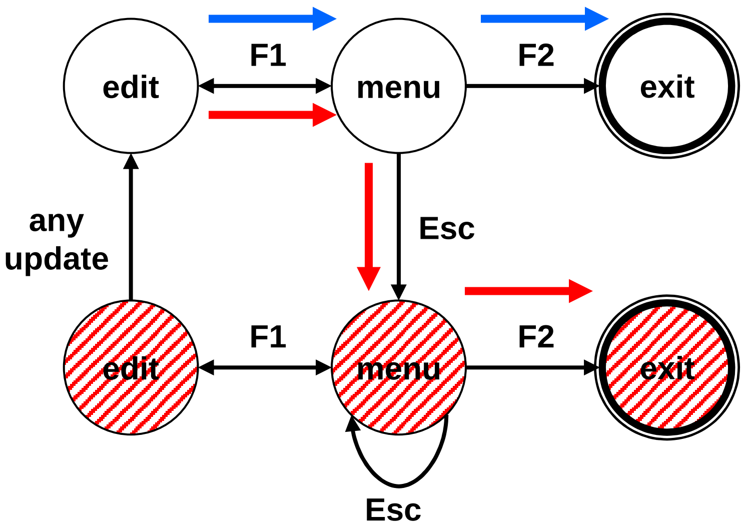
- New keyboard layout



intend F1-F2 (save)

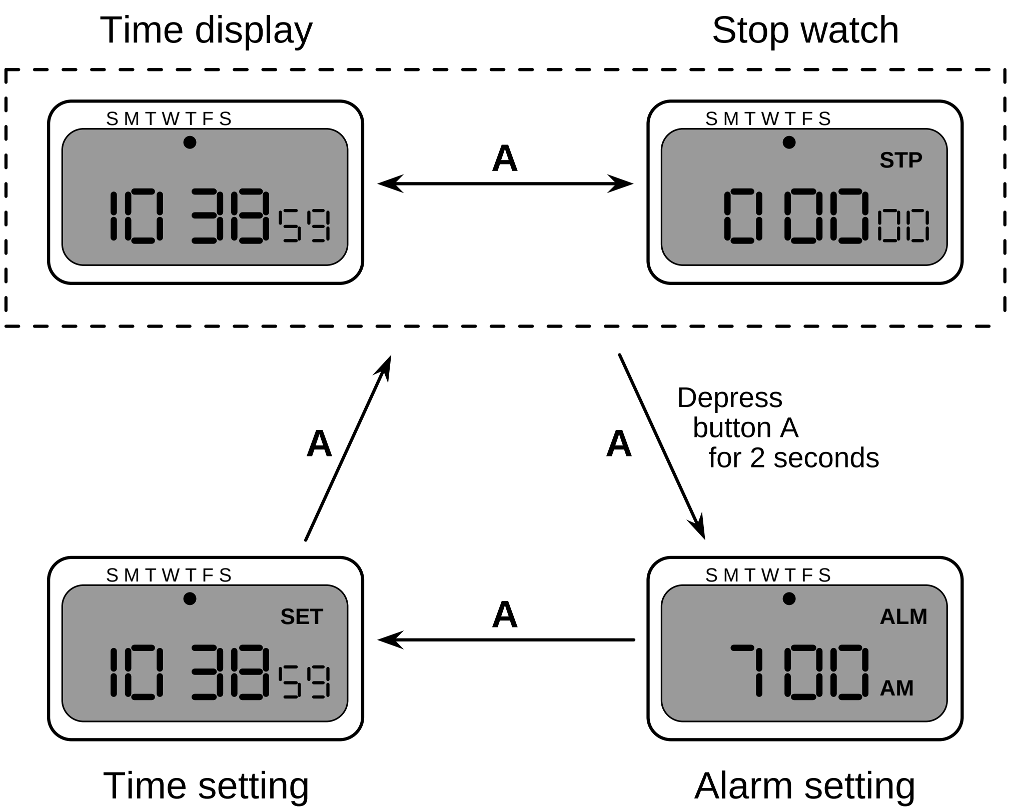
finger catches Esc

F1-Esc-F2 - disaster!



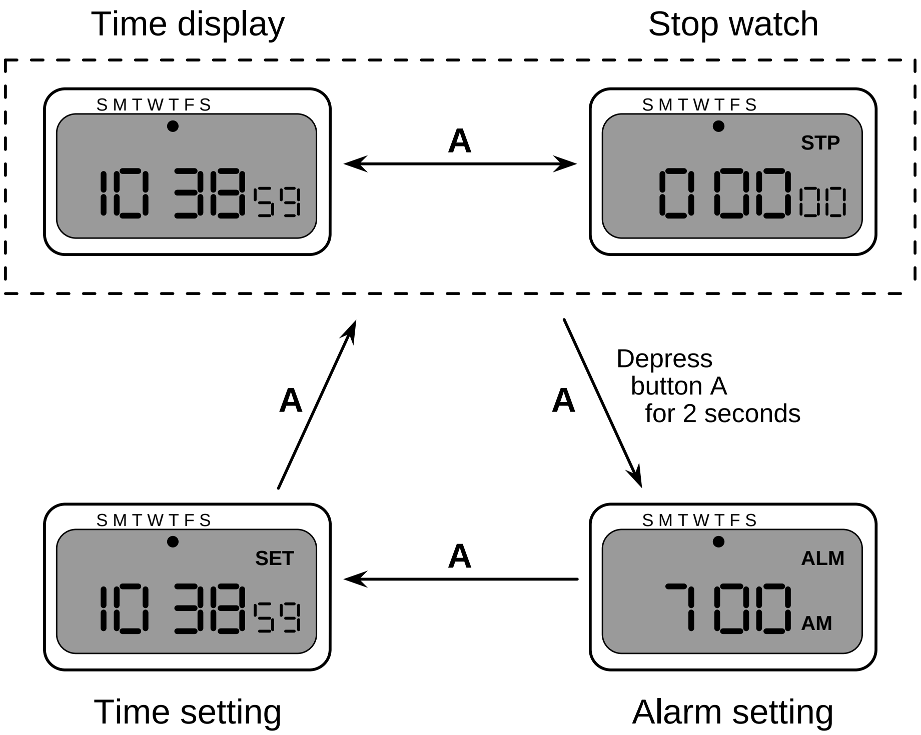
# Digital Watch – User Instructions

- Different modes
- Limited interface
  - 3 buttons
- Button A
  changes mode



# Digital Watch – User Instructions

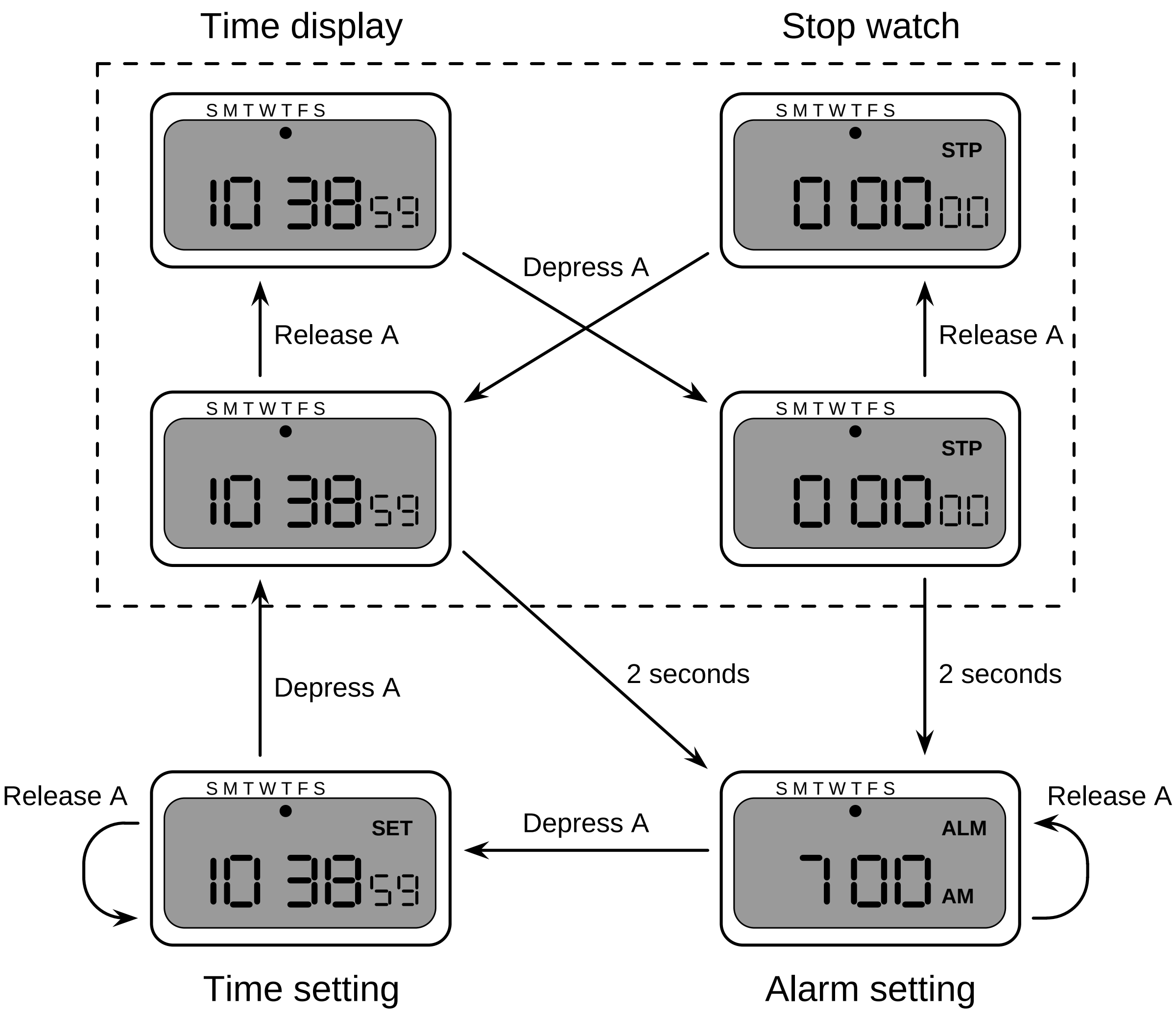
- Dangerous states
  - *Guarded*
    … by two second hold
- Completeness
  - Distinguish depress A and release A
  - What do they do in all modes?



# Digital Watch – Designer Instructions

and ...

that's just
one button



# Textual Dialog Notations

# Grammar – An English Grammar

A sentence is a noun phrase, a verb, and a noun phrase.

<*S*> ::= <*NP*> <*V*> <*NP*>

A noun phrase is an article and a noun.

<*NP*> ::= <*A*> <*N*>

A verb is…

<*V*> ::= **likes** | **hates** | **eats**

An article is…

<*A*> ::= **a** | **the**

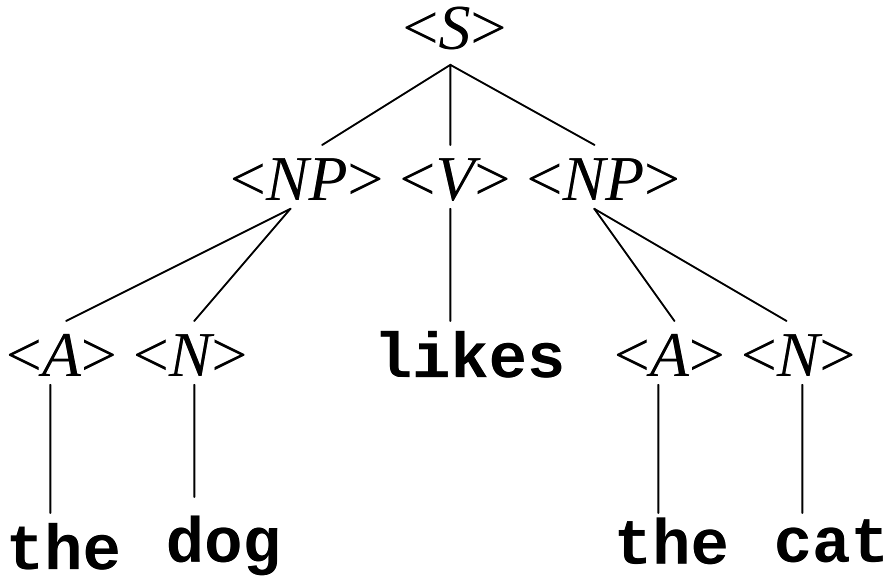
A noun is...

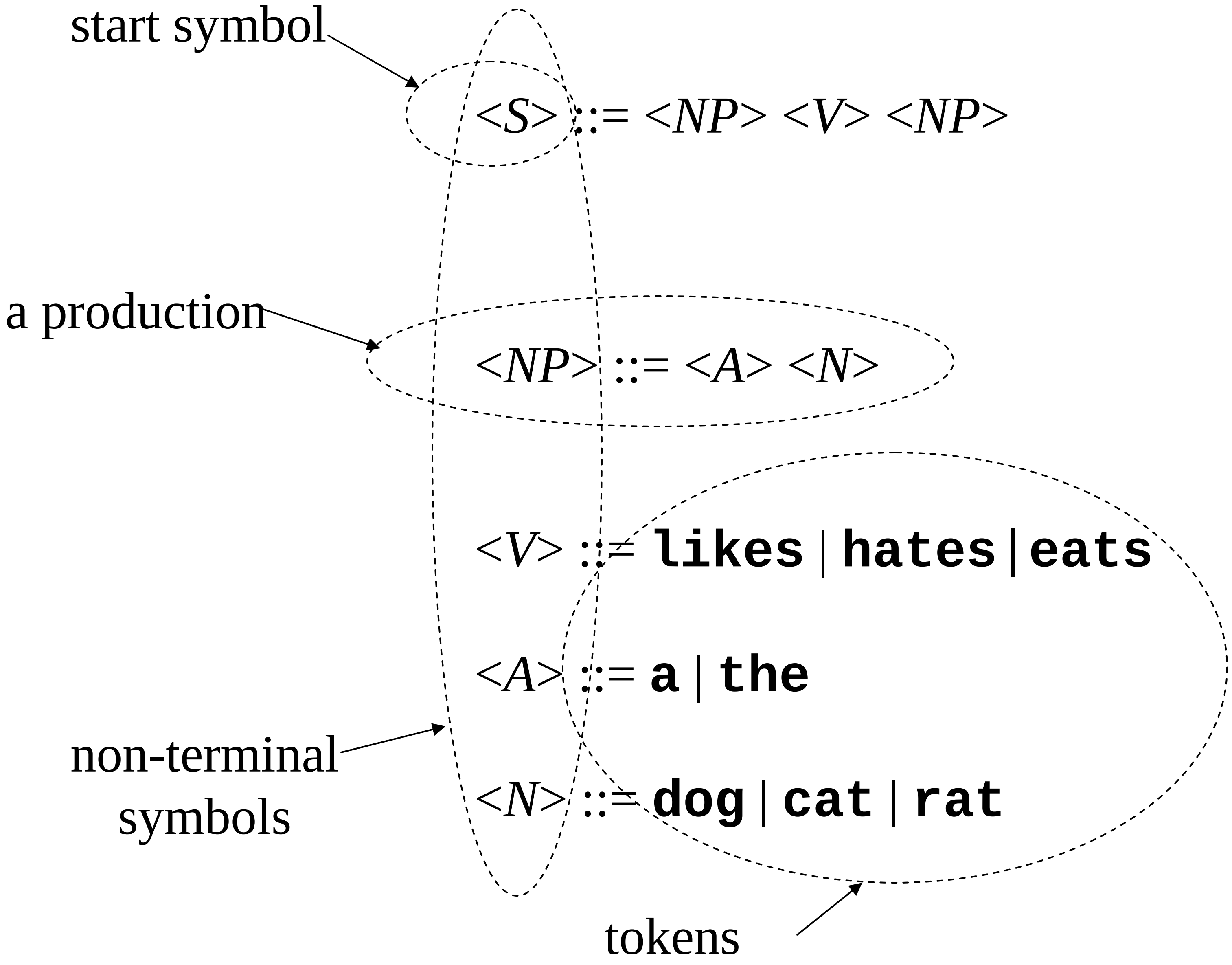
<*N*> ::= **dog** | **cat** | **rat**

# How the Grammar Works

- The grammar is a set of rules that say how to build a tree – a *parse tree*
- You put <*S*> at the root of the tree
- The grammar’s rules say how children can be added at any point in the tree
- For instance, the rule says you can add nodes <*NP*>, <*V*>, and <*NP*>, in that order, as children of <*S*>

# A Parse Tree





start symbol

<*S*> ::= <*NP*> <*V*> <*NP*>

a production

<*NP*> ::= <*A*> <*N*>

<*V*> ::= **likes** | **hates** | **eats**

<*A*> ::= **a** | **the**

non-terminal symbols

<*N*> ::= **dog** | **cat** | **rat**

tokens

# BNF Grammar Definition

- A BNF grammar consists of four parts:
  - The set of *tokens*
  - The set of *non-terminal symbols*
  - The *start symbol*
  - The set of *productions*

# BNF Grammar

- Tokens
  - Smallest units of syntax
  - They are atomic: not treated as being composed from smaller parts
- Non-terminal symbols
  - Stand for larger pieces of syntax
  - They are strings enclosed in angle brackets, as in <*NP*>
  - The grammar says how they can be expanded into strings of tokens

# BNF Grammar

- Start symbol
  - The particular non-terminal that forms the root of any parse tree for the grammar
- Productions
  - The tree-building rules
  - Each one has a left-hand side, the separator **::=**, and a right-hand side
  - The left-hand side is a single non-terminal
  - The right-hand side is a sequence of one or more things, each of which can be either a token or a non-terminal

# BNF Grammar

- Alternatives
  - When there is more than one production with the same left-hand side, an abbreviated form can be used
  - The BNF grammar can give the left-hand side, the separator **::=**, and then a list of possible right-hand sides separated by the special symbol |

# Example

<exp> ::= <exp> + <exp> | <exp> * <exp> | ( <exp> )
| a | b | c

Note that there are six productions in this grammar.
It is equivalent to this one:

<exp> ::= <exp> + <exp>
<exp> ::= <exp> * <exp>
<exp> ::= ( <exp> )
<exp> ::= a
<exp> ::= b
<exp> ::= c

# Parse Trees

- To build a parse tree, put the start symbol at the root
- Add children to every non-terminal, *following any one of the productions for that non-terminal in the grammar*
- Done when all the leaves are tokens
- Read off leaves from left to right—that is the string derived by the tree

# Practice Exercises

*<exp>* ::= *<exp>* **+** *<exp>* | *<exp>* * *<exp>* | **(** *<exp>* **)**
| **a** | **b** | **c**

Show a parse tree for each of these strings:

```
a+b
a*b+c
(a+b)
(a+(b))
```

# Compiler Note

- What we just did is *parsing*: trying to find a parse tree for a given string
- That's what compilers do for every program you try to compile: try to build a parse tree for your program, using the grammar for whatever language you used
- Details – Course Compiler Construction

# Constructing Grammars

- Example: the language of Java declarations: a type name, a list of variables separated by commas, and a semicolon
- Each variable can be followed by an initializer:

```
float a;
boolean a,b,c;
int a=1, b, c=2;
```

# Constructing Grammar

- Easy if we postpone defining the comma-separated list of variables with initializers:

```
<var-dec> ::= <type-name> <declarator-list> ;
```

- Primitive type names are easy enough :

```
<type-name> ::= boolean | byte | short | int
              | long | char | float | double
```

# Constructing Grammar

- That leaves the comma-separated list of variables with initializers
- Again, postpone defining variables with initializers, and just do the comma-separated list part:

```
<declarator-list>  ::=  <declarator>
                      |  <declarator>  ,  <declarator-list>
```

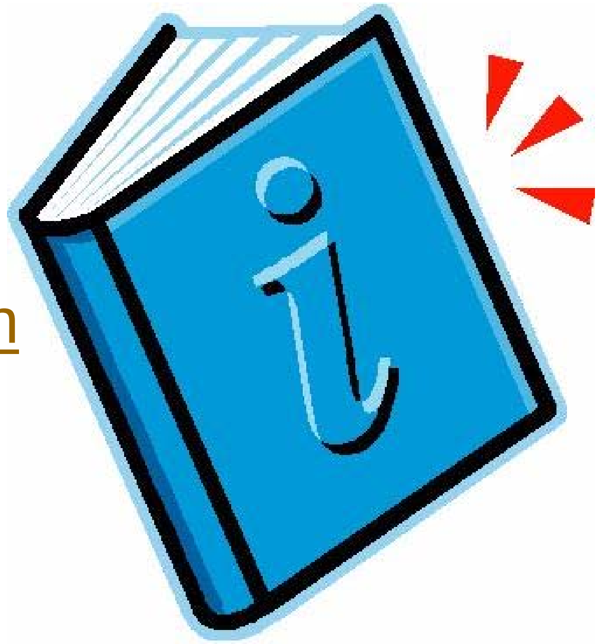
# Constructing Grammar

- That leaves the variables with initializers:

```
<declarator>  ::=  <variable-name>
                   |  <variable-name>  =  <expr>
```

- Definitions for <*variable-name*> and <*expr*> and much more …

# References



- Chapter 16 - Human Computer Interaction by Dix et al.
- Interactive Tutorials on Petri Nets, Wil van der Aalst, et al. TU Eindhoven, Netherlands
- Modeling and Simulation, P. Fishwick