

Inside Chisel

- design goals
 - in the style of Unix
 - command line only
 - one command to achieve one task well
- chisel is a package with at least three command line programs
 - txt2pen convert a txt file into a pen file
 - pen2map convert a pen file into a map file (doom3)
 - rndpen generate a random pen file
- for your coursework you should consider extending:
 - pen2map or txt2pen or introducing a third which could manipulate a pen or txt file

rndpen

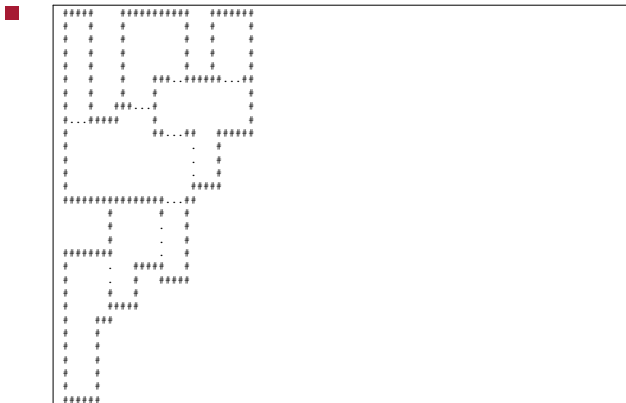
- generate a random pen map
- highly alpha code, but it will generate a corridor based random pen file
- the program does always find a map
- so some experimentation is required for the pseudo random numbers to mesh with the algorithm to generate a map

rndpen

- ```
$ rndmap -h
Usage rndpen [-a minroomsize] [-b maxroomsize] \
 [-c mincorridorlength] [-d maxcorridorlength] \
 [-e totalcorridorlength] [-h] [-o outputfile] \
 [-s seed] [-x maxx] [-y maxy]
-a minroomsize (default is 6)
-b maxroomsize (default is 13)
-c mincorridorsize (default is 15)
-d maxcorridorsize (default is 70)
-e totalcorridorlength (default is 300)
-o outputfile (default is stdout)
-s seed (default is 3)
-x minx for whole map (default is 120)
-y maxy for whole map (default is 80)
```

## rndpen

- ```
$ rndmap -s 3 -a 5 -b 10 -c 5 -d 10 -e 20 -x 30 -y 30 | p
```
- notice how the command line tools can be combined using the pipe

rndpen**rndpen**

- rndpen prioritises placing random corridors on the map
- it then tries to fill in the remaining gaps with boxes and will combine boxes to give rooms of desired min/max dimensions
- it also restricts the number of walls to 8
- it might be useful if you wanted to generate a map quickly
 - however it might generate concave rooms (pen2map can only encode convex rooms currently)

Inside: txt2pen

- source is in one file:
 - \$HOME/Sandpit/chisel/python/txt2pen.py
 - 690 lines of Python
- uses the following command line options

```

$ cd $HOME/Sandpit/chisel/python
$ python txt2pen.py -h
-d debugging
-h help
-V verbose
-v version
-o outputfile name

```

Inside: txt2pen

- notice the `-o` option which takes an additional argument (filename)
- it uses the `getopt` module to handle the options
 - see function `handleOptions`

Inside: txt2pen

```

def handleOptions ():
    global debugging, verbose, outputName

    outputName = None
    try:
        optlist, l = getopt.getopt(sys.argv[1:], 'dho:vV')
        for opt in optlist:
            if opt[0] == '-d':
                debugging = True
            elif opt[0] == '-h':
                usage (0)
            elif opt[0] == '-o':
                outputName = opt[1]
            elif opt[0] == '-v':
                printf ("txt2pen version " + str (versionNumber) + "\n")
                sys.exit (0)
            elif opt[0] == '-V':
                verbose = True
        if l != []:
            return (l[0], outputName)
    except getopt.GetoptError:
        usage (1)
    return (None, outputName)

```

- it uses a dictionary to maintain the defines
- stores the map in a 2D list (array)
 - mapGrid

Inside: txt2pen**Inside: txt2pen**

- it determines the walls of a room
 - it finds the room number (location)
 - moves to the top left inside the room (generateRoom)
 - it then attempts to turn left as it moves around the room (the wall is always on the left)
 - examine scanRoom for the implementation
 - it looks the square forward and square forward left comparing the two characters: ## or -- or #-
 - # wall and - for space
- a space should be thought of as not a wall

Inside: txt2pen

- scanRoom will start at the top right corner of a room and walk around the edge with the wall always on the left
 - it builds a list of walls, a wall stops/starts at each turn
- if it sees ## then it must turn right
 - the old wall is stored and a new start position is remembered
- if it sees -- then it must turn left
 - the old wall is stored and a new start position is remembered
- if it sees #- then it continues moving a square forward

Extending chisel (txt2pen)

- one of the obvious improvements is for chisel to automatically introduce lights
 - add another option to enable automatic lighting
 - -l
- copy scanRoom into a new function introduceLights
- adapt this new function to add lights
 - but only if the rooms has no user defined lights

Inside: pen2map

- chisel/python/pen2map.py is 2086 lines of Python

- ```
$ cd $HOME/Sandpit/chisel/python
$ python pen2map.py -h
Usage: pen2map [-c filename.ss] [-dhmtvV] [-o outputfile]
-c filename.ss use filename.ss as the defaults for t
-d debugging
-e provide comments in the map file
-g type game type. The type must be 'single'
-h help
-m create a doom3 map file from the pen
-s generate statistics about the map fil
-t create a txt file from the pen file
-V generate verbose information
-v print the version
-o outputfile place output into outputfile
```

## Example style sheet for the map

- how are textures defined - could use the defaults - and ignore this slide!
  - or examine chisel/python/tiny.ss

- ```
# style sheet for simple doom3 maps
define floor textures/hell/qfloor
define portal textures/editor/visportal
define open textures/editor/visportal
define closed textures/hell/wood1
define secret textures/hell/bricks1a_d
define wall textures/hell/cbrick2b
define ceiling textures/hell/wood1
```

pen2map

- reads in a pen file and converts it into a doom3 map file
- the pen map is parsed by a top down recursive descent parser
 - the pen syntax is described by an ebnf grammar
 - (extended backus naur form)
 - hand translated into a top down recursive descent parser
- recursive descent parsers are fast and straightforward to implement once the grammar is defined
 - they also allow for strict syntax checking of input
 - they are used extensively in the construction of compilers

ebnf

- consists of terminal symbols and non-terminal production rules which define the legal sequence of symbols

- in C++ for example, a terminal symbol might be while, for, do, = ; 0 etc

- a rule might be:

```
assignment := lhs "=" rhs =:
```

- meaning the assignment rule is satisfied if there is a legal lhs followed by = followed by rhs

pen example

- ```
ROOM 1
 WALL
 1 21 18 21
 18 21 18 14
 18 14 1 14
 1 14 1 21
 DOOR 18 18 18 17 STATUS OPEN LEADS TO 2
 MONSTER python_doommarine_mp AT 13 18
 LIGHT AT 12 20
 LIGHT AT 4 15
 LIGHT AT 15 15
 SPAWN PLAYER AT 3 18
END
```

**pen grammar in ebnf**

- ```
FileUnit := RoomDesc { RoomDesc } [ RandomTreasure ] "END"
RoomDesc := 'ROOM' Integer
           { WallDesc | DoorDesc | TreasureDesc } 'END' :
WallDesc := 'WALL' WallCoords { WallCoords } =:
WallCoords := Integer Integer Integer Integer =:
DoorDesc := 'DOOR' DoorCoords { DoorCoords } =:
```

pen grammar in ebnf

- ```
DoorCoords := Integer Integer Integer Integer Status
 'LEADS' 'TO' Integer =:
Status := 'STATUS' ('OPEN'
 | 'CLOSED'
 | 'SECRET'
) =:
TreasureDesc := 'TREASURE' 'AT' Integer Integer
 'IS' Integer =:
RandomTreasure := 'RANDOMIZE' 'TREASURE' Integer
 { Integer } =:
```

## ebnf meta symbols

- `{ foo }`
  - means it is legal to have 0 or more occurrences of `foo`
- `[ foo ]`
  - means it is legal to have 0 or 1 occurrence of `foo`
- `( foo | bar )`
  - here the `(` and `)` group together the extent of the `|`
- `"foo"` represents the terminal symbol `f o o`
- as opposed to the rule `foo`

## Translating ebnf grammar into a top down parser

- once the grammar is defined it is straightforward to implement a top down parser
- if the grammar is said to be well formed if we only need to look at the next token to determine the flow of control in the parser

## Translating ebnf grammar into a top down parser

- we define a few helper functions
  - `expect ("foo")` insists that the next token is "foo" and generates an error if it is not "foo"
  - if "foo" is seen the consume this symbol and move onto the next
- `expecting (list)`
  - returns `True` if any symbol in `list` matches the current symbol
- `integer` return `True` if the current symbol is an integer
  - if `True` store the value of the integer in `curinteger`