SEPT 2, 2009 COMP 302 PAUL HUSSMAN

- Imperative Languages - You tell it exactly what to do
  - procedural languages: C, Fortran, Cobol, Java
  - Object Oriented Languages: Objects bring together data and operations. eg: Java, C++, smalltalk

- Declarative Languages - You tell it what you want to be true. (what it should look like in the end)
  - its up to the machine to make it happen.

Example: GCD (a,b)

algorithm (return to top):
- a==b ? → answer a.
- a>b replace a with a-b
- a<b replace b with b-a

typical code

```
int gcd (int a, int b) {
    while (a != b) {
        if (a>b)
            a = a-b;
        else
            b = b-a;
    }
    return a;
}
```

- Functional Languages (declarative) ← uses recursion instead of loops.
  - functions created dynamically and passed around.
  - "pure environment" - no memory change, (no state change) from execution
    - input → output.

Example in Scheme (lambda ← creates a new function)

```
(define gcd (lambda (a b)
    (cond ((= a b) a)
          ((> a b) (gcd (- a b) b))
          (else (gcd (- b a) a)))))
```

- Prolog (language that is fully declarative) not used so much since not efficient these days.

```
GCD(A,B,G):- A=B, A=G
GCD(A,B,G):- A>B,
             C is A-B
             GCD(C,B,G)
GCD(A,B,G):- B>A
             C is B-A
             GCD(C,A,G)
```

- What does a language NEED?
  - sequencing
  - conditionals
  - iteration/recursion (looping)

COMP 302 Sept. 4, 2009 (3rd class) PAUL HUSSMAN,

Assignments: make language in javascript – implementing a wiki-markup language.

- wiki's don't use HTML, its more readable
- templates allow you to do macro-expansions → we will build a complete functional language

HTML – simple minimal webpage template: → <html> <head> </head> <body> </body> </html> – this is what we care about,

- Inside the <body> we care about: <p> </p> paragraphs
  -     unordered list
  -     ordered list.
  - <dl> <dd> </dd> </dl> definition list

  also images, urls, bold, italics, pre, div, tables
- won't use many interactive things (POST, GET, forms...)

JAVASCRIPT

- we will focus on a java-like subset for programming.
- Javascript has NOTHING to do with Java. It was originally called LIVESCRIPT. Java was becoming popular, so they called it Javascript as a marketing tactic.
- It is a language that runs in webpages → browser has interpreter for Javascript. Browser is the execution environment.
- considered a SCRIPTING language (means that it is interpreted).
- it is typeless (unlike Java) → we dont write "int", "float", etc.
- Variables can be of any type at any point. The interpreter worries about it for us,
- Javascript will typically be embedded inside a webpage → within HTML.
- Html special tag <script> [javascript program] </script>

  eg: <script language="javascript">
  or <script type="text/javascript"> ← most common.
  or <script type="application/javascript">

  – For the most part, the various browsers will except either forms.

  make sure stuff works in FIREFOX

- What can we have inside javascript?
  - we have all our usual (Java-like) control construct eg: while, for, do, if, switch, variable assignment, arithmetic assignment, boolean expressions.
  - also we have the HTML stuff. It can refer to the rest of the webpage that its in. It uses DOM (Document Object Model), and this is how we refer to the rest of the webpage. – we don't need to worry much about this.

```
<script language="javascript">
var f=1;
for (var i=1; i<10; i++) {
    f *= i;
    document.write(i+'! = '+f+'<br/>');
}
</script>
```

} goes in <body>

- You could also just put 'javascript:' into the address bar of a browser.
  eg: javascript: 142 % 8
- You can also use the Error Console → works there too.
- For assignments we need to write out the whole webpage, but these are good testing methods.

JAVASCRIPT SYNTAX

DATA - all variables are declared by **var**, and later they take on the form of the type.

types: 3 basic types.
- Number, 32-bit ints or 64-bit floats, but javascript takes care of conversions.
- String 'foo' or "foo"
- Boolean true/false.

COMP 802   SEPT 9 2009   PAUL HUSSMAN.

last time : Javascript — embedded in webpages, prog lang, DOM
— write pages — java-like

- math.sqrt() — only a few TYPES in JS.
- math.floor   ↳ var.
- .random
- .ceil

- type conversions; JS will convert vars to appropriate type

var i = 0;   for (i=0; i<10, i++){

- what if we have string/numbers?

var x = "0"

x = x *7 — *7 needs a number

Number → string

var x = 0

var s = x; // doesn't cause any conversions

API s = string(x)

s = "" + x ← "7" + x = 70 — people do this a lot.

↳ s == "0"

- other ways

s = x.toString();

String → Number

var s = "7";

var x = Number(s);

= ~~s + 0~~ // this will just put a 0 on the end.

= s - 0 // subtract nothing.
↖ causes a conversion to a number

s.parseInt();
s.parseFloat();   } both return NUMBERS

s = "7 java"   s.parseInt(); // extracts the number & converts it; returns 7
s.parseFloat(); // returns 7.0.

09/09/09 CONT'D.

BOOLEANS
→ true, false.
→ weird conversions!

eg: if("hello")

String	Boolean	Number
"hello" →	true (any non-empty string)	← anything but Ø & NaN
" " →	false (empty string)	← Ø, NaN (Not a number)

ex:
```
var b = false;
var s = b + ""; // returns s = "false"
```

★ source of interesting problems!
ex: ① boolean false / ② string "false" / ③ boolean true — can't make the round trip.

END of Javascript Basic Types.

other things we need to know.

FUNCTIONS:

```
function foo(x) {      // note: don't need to specify return type. or arg type
    return 17;
}    ← converted or discarded automatically
```

ex:
```
function foo() {}
var x = foo();    // will not be NULL, but similar.
                  // x will be undefined. since it was never defined. interchangable w/ NULL.
                  // infact NULL == undefined.
```
(NULL ← value, undefined ← state, but considered equivalent)

Javascript equality

= } same as java.
== } → tests inequality and does type conversions including NULL/undefined.
=== } this one will say undefined === Null is FALSE. (will never need this).

- functions can also be assigned to variables.

```
var f = function(x) {...}    //here the function has no name, but you use f to call it.
```
-or-
```
var f = Function("x", "return x+x;");    // we define in terms of strings
                                         - this allows for dynamic functions
```
eg: f(3) → 6
f("3") → 33

09/09/09 CONT'D.

OBJECTS

→ in JAVASCRIPT nearly everything is called an Associative Array.

ASSOCIATIVE ARRAY : LIST OF KEY/VALUE pairs.

eg: var x = { Key1 : value1, key2 : value2 };

= { k_1:v_1, k_2:v_2 }

x.Key1 = 17;

x.Key2 = "hello";

var x = {}; // object w/ no fields.

x.key1 = "value"; //now it has a field.

var q = x.key5 ; // q will be 'undefined'; but it can be used later.

delete keyword allows removal of fields.

ASSIGNMENT #1 handed out. → due sept 21

SEPT 11, 2009 COMP 302 PAUL HUSSMAN.

last Time → conversions, objects (associative array (key-value pairs)), functions [→ Uppercase F means strings.]

Arrays

```
var a = [3, "hello", 7.2];   //dynamically expandable
    a[3] = "Whatsup";
    a[8] = 6;
    if (a.length) == 9) ...          ← (spot for next element)
```

make an array one longer → a[a.length] = ...

You can add fields that aren't even indices.

```
a.foo = 18;   //doesn't affect a.length.
a.setAttribute("foo", 7)  // same as a.foo = 7;
a.getAttribute("foo")
var o = {foo: "bar"};
    o["foo"]   // == "bar";
```

in

```
if("foo" in o) {...}     // asks if there is a "foo" in o.
var o = {foo: "bar", x:7};
for (var y in o) {
    document.write("field " + y + " = " + o[y]);
}
```

★ bad parts of javascript.

→ optional semicolons ; ⇐ WE MUST USE THEM

return true; vs. return
true; } Not the same! one's true, other is false

be careful w/ return statements and break, continue keywords.

→ vars do not have to be declared before being written.

eg: f = hello;
var f = hello; } slightly different meaning.
↳ current local scope. Otherwise GLOBAL.

- example related to assignment → computer

good free debugger → plugin for Firefox called Firebug

in lab → its in usr/local/plugs/firebug-1.4.2.xpi

lets you debug javascript & html.

enough of JS.

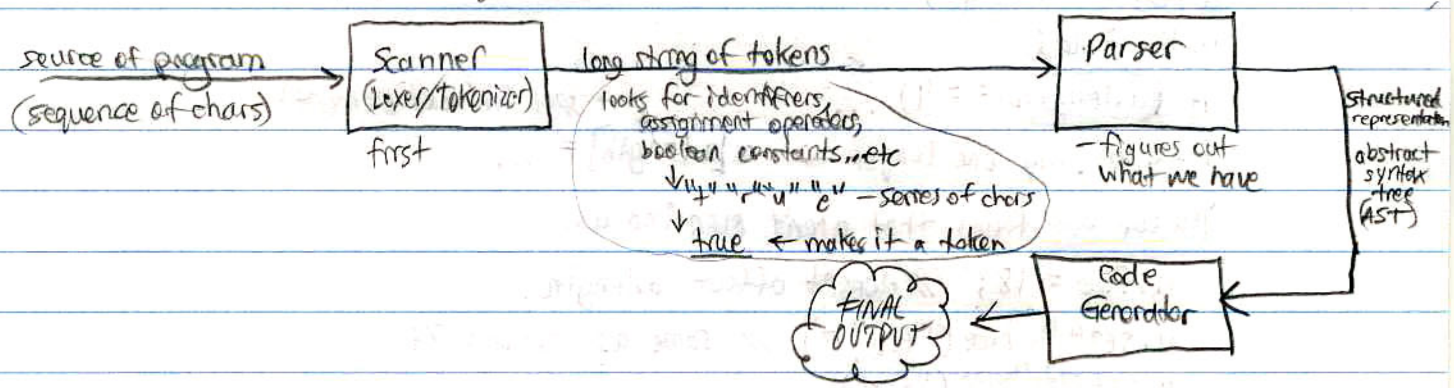
Now: COURSE MATERIAL

Develop a language.

↳ need to specify → precisely → hierarchy of grammer

one formalism defines simple structures,

a different formulation for larger structures



monday
SEPT 14 2009 COMP 302 PAUL HOSSMAN

- switch statements in javascript is more flexible

switch (v) { // variable is ANY TYPE

case (arbitrary expression) : ← can be real code (not suggested)

break;

* goes 1 by 1.

scanning (cont'd)

source → Scanner → tokens → Parser → Code generator

Scanner: handles simple constructs; Parser: more complex constructs

in a for-loop → for (var i = 0; i < 10; i++) {

SCANNER SAYS → keyword, assignment, number, white space.

↳ takes stream of chars and makes stream of code.

- if you the letters f o r, you have to check what's before and after it → ∴ need to recognize patterns of characters.

★ to recognize sequences/patterns of chars, we use <u>REGULAR expressions.</u>

REGULAR EXPRESSIONS (RE's)

↳ description language

definition → A Regular Expression is...

1) a single character : a
2) an empty string : ε
3) Conjunction: if R1 is a RE and R2 is a RE, then R1R2 is a RE } contatenation of 2 RE's is an RE
4) Disjunction: if R1 is a RE and R2 is a RE then $R_1 | R_2$ is a RE } R_1 OR R_2 is a RE.
5) If R is a RE, then R* is a RE } any # of copies of R is a RE. (including ∅) [Kleen-star]

example: for indentification

it can start with a letter, an underscore...

after that you can have an arbitrary number of letters, underscores, digits...

in terms of regular expressions: A|B|C|D.... |Z|a|b|c|d.... |z|0|1|2|3 ... 9|_|$|...

so identifiers can be A-Z, a-z, _, $, (chars) 0-9 (digits) ⇒ character (character | digit)*

this is tedious → in practice there are character classes... ↗

Hilroy

a RANGE can be given: [A-Z] ≡ any char in A through Z

eg: [A-Z a-z _ $] ⇐ this is a class for all starting chars.

so for identifiers you can have a regex like this:

[A-Z a-z _ $][A-Z a-z _ $ 0-9]*

inverse classes (everything BUT the following).

[^A-Z] ≡ everything BUT range of A-Z (capitals).

↑ negate

- But how do you get a '–' character: put it as the 1st char [–...]
- A-Z a-z shows up all the time, so there are built-in char classes.
→ they are specified seperately by special characters.

eg: \n ⇝ newline, \t ⇝ tab, \w ⇝ ANY WORD character, \d ⇝ any digit

\s ⇝ white space, ...

→ . ⇝ any character other than a newline.

→ $ ⇝ end of the line

→ ^ ⇝ (outside of a class) beginning of the line

eg: entire line: ^.*$ (from beginning including any character as many times as they exist, to end)

Note RE's match as long as possible! eg: a* will match: [aaaaaaa]b

↑ longest possible iteration.

→ some systems allow you to NOT do the longest match → special operation in JS.

→ + ⇝ at least one in a * match

eg: a* matches ε, a, aa, aaa, aaaa, ⋮   a+ matches a, aa, aaa, aaaa, ⋮ (No empty string)

→ ? ⇝ 0 or 1 copies of something

eg: a? matches ε|a.

→ a{2,6} ⇝ matches aa, aaa, aaaa, aaaaa, aaaaaa [2 ↓ 6] ⟶ specifies a range

SUB-PATTERNS

→ typically people put brackets around stuff to indicate what they're doing. FOR CLARITY.

→ brackets can also go around chunks that can be referred to later on!

eg: a(bc)d ⇐ can be refferred to later as \1

ab(cd)efgh(ijk)lmn

\1 (cd), \2 (ijk)

- allows us to do some neat things. → "string", 'string', "string' ⇐ doesn't work

to describe a string ["'][^"']*["'] allows the bad case ⤴

(["']) [^"']* ["'] ⟶ (["']) [^"']* \1 will not allow the bad case

→ in JavaScript RE's are built in.

String.match(/RE/);

"hello".match(/el*/); // returns array of matches (or NULL if no match)

SEPT 16, 2009 COMP 302 PAUL HUSSMAN.

- last time: regular expressions

\w ⇝ word char	\b ⇝ word boundary	regular expression extentions
\s ⇝ space/whitespace	\B ⇝ non-word boundary	
\d ⇝ digit	(...) \1 ⇝ refer to part of RE	↳ see last lecture above

SCANNING

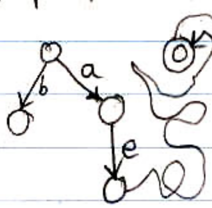
- use Regular expressions to construct TOKENS.
- converts characters to tokens.

TWO TECHNIQUES of doing this...

① FORMAL APPROACH

- can be done by hand, but a lot easier w/ a tool
- will convert our RE's into something easily processed: N.F.A. [nondeterministic finite automaton]

NFA: graph w/ two kinds of nodes: o, ◎, representing states.



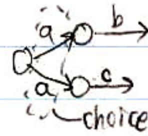
move through graph based on what characters we see

if the accepting state/final state is reached, then the regular expression is recognized

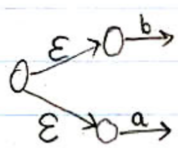
OTHERWISE its rejected.

what does NON-DETERMINISTIC mean?

- it kind of knows what will happen ahead of time.

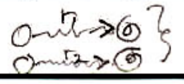


if looking for "ab..." then it won't get stuck on the 'c', instead it will find the path that works (if any).

- 

- we also have 'ε' meaning empty string/character
- ★ these transitions can happen at any point, arbitrarily.

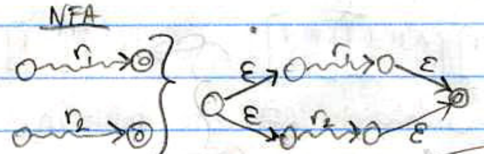
- converting regular expressions to NFA's

	RE	NFA	
1)	a	o —a→ ◎	(always one start state & one accepting state)
2)	ε (empty string)	o —ε→ ◎	
3)	$r_1 r_2$		concatenation of 2 RE's r_1 & r_2.

PAUL HUSMAN

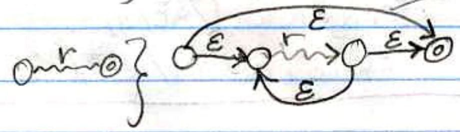
RE | NFA

4) $r_1 | r_2$



for the ∅th case

5) r^* (∅ or more repetitions of r)



examples: /* ...... */

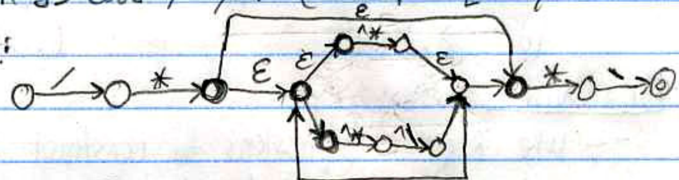
what do we allow inside this type of comment?

/*([^*]|*[^/])**/

↑ incorrect

in JS code: /\/*([^*]|*[^/])**\// (note escape chars)

CORRESPONDING NFA:



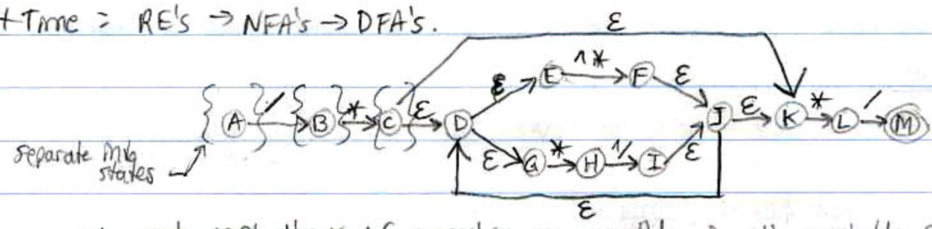
this is rather slow, since its non deterministic.

DFA: deterministic finite automaton → eliminates choice.

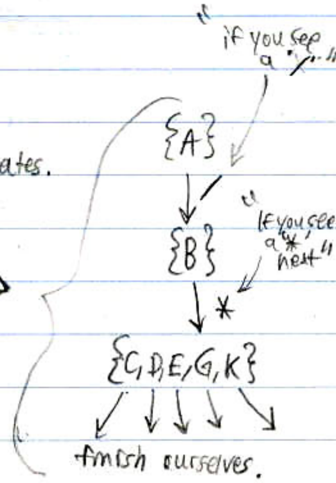
NFAs can be converted to DFAs by the SET OF SUBSETS approach.

COMP 302 SEPT 18, 2009 PAUL HUSSMAN

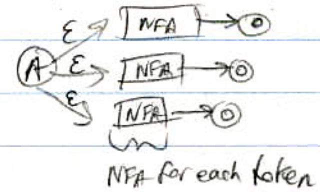
Last Time : RE's → NFA's → DFA's.



- at each part think of ourselves as possibly in all reachable states.
  - we end up with another Finite Automaton, but now we do not have any ε symbols →
- this would be a very small DFA, but usually/often that's not the case, and there are tools to do it for you which is much less error prone.
  eg: of tools: LEX or FLEX



- So for each token we need a NFA (or DFA.) so we start in a particular state, A, and and reach the final state through a NFA.



- a cheap/trivial scanner using RE's directly is just a bunch of if-statements to find the string that matches a token. eg ⇒

```
function scan(str) {
  if (str.match(/.../)) {
  } else if ____
  } else if ____
  } else ...
```

- good idea: var value;

```
value = str.match(/.../);
  if (value) {
    return token TOKEN_ID
      value : value[0];
  }
```

// this helps keep track of them.

- extra complication: should find the longest match, but we can just test for ones that match the longest ones first.
  Alternatively you could test all and keep the longest one (wasteful).
- No Match? - What do you return if nothing matches?
  ☆ NEED a DEFAULT case → returns a single character.
- Context: we also need to look for/think about context. Eg: inside a string we might find tokens, but they don't count!
  bottom line: In practice we may not need all kinds of tokens in all contexts.
  NEED TO TELL THE SCANNER WHAT TOKENS YOU CARE ABOUT.

```
function scan(str, tokens) {
  if (tokens[TOKENS_ID] && str.match(/.../)) { ...
```

→ which tokens do we care about within this context

Hilroy

once we have our scanner, we are ready to make it into a language.

PARSING

→ several ways of doing this. We will look at: - simple method
- more complex method,

→ We now have a stream of tokens
we need to read them in and build higher level structures.

→ a grammar is used to describe the higher-level structures.
↳ set of rules defining how to construct higher-level structures from lower level structures (tokens).

→ rules work like this:

rulename → piece1 piece2 piece3

rulename: higher level structure ← this is called the non-terminal

piece1 piece2 piece3: pieces that make up the higher level structure ← here we have:
- other rule names (also non-terminals)
- symbols eg: '=' (tokens)

(NOTE syntax varies)

we also have choice (more than one way to produce a rule).

rule name → ______ | ______ | ______

→ now we need to use these rules to make a higher-level structures

function declaration → TOKEN_FN TOKEN_ID TOKEN_LB args TOKEN_RB TOKEN_LBRACE stmtlist TOKEN_RBRACE

(TOKEN_LB = left bracket, TOKEN_RB = right bracket, TOKEN_LBRACE = {)

note: white space matters after keyword

args → ______

stmtlist → ______

BNF Grammar was invented by Backus-Naur Form

people mostly use EBNF grammars (extended BNF grammars).

if something doesn't have to be there you can do fname → TOKEN_ID | ε

but its easier to do it right in the grammar eg: TOKEN_ID ? ← signifies 0 or 1 copies.

- what if you want many arguements → we need to have the rule extensable.

→ args → ε | id | arglist

arglist → TOKEN_ID TOKEN_COMMA arglist | ε

} recursively lets it be long enough

note: this version leaves a comma and isnt perfect but it shows that we use recursion to make LISTS.

In EBNF we can use * to specify a list

(TOKEN_ID TOKEN_COMMA)*

COMP 302 SEPT 21, 2009 PAUL HUSSMAN

Last time → finished scanning → we get tokens
→ parsing → build larger structures internally

- we need to specify our language structure → with EBNF grammar (+ ? *)
which is made up of rules which have non-terminals & terminals.

We will stick to restrictions for the rules

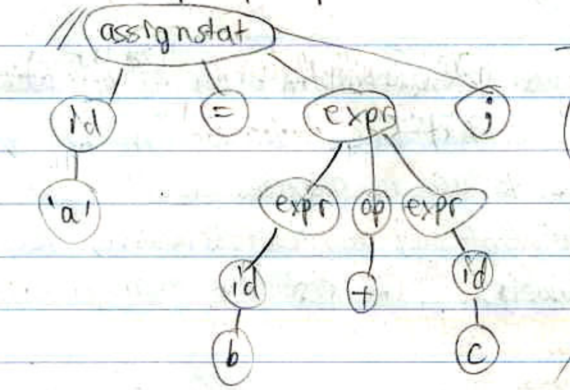
- left hand side of each rule is a single symbol (or name) (Context-free Grammar)

Assignment Statement: a = b + c ;

assignstat → id '=' expr ';'

expr → id | expr op expr

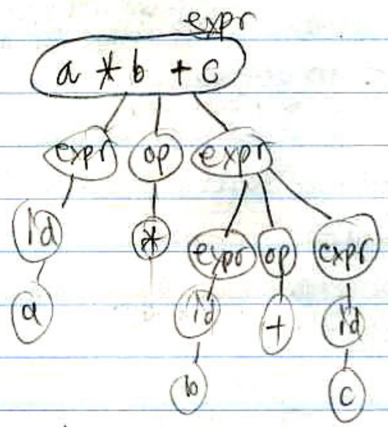
a = b + c ;



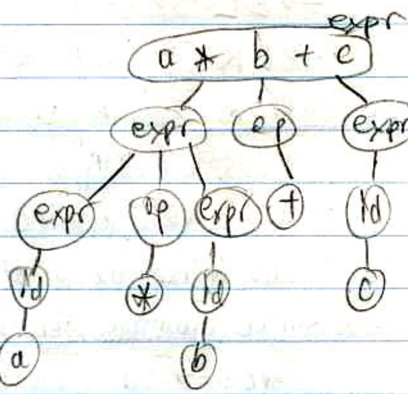
Parse tree for our input & given grammar

this is a Concrete syntax tree
↓
similar to Abstract syntax tree

note: order of operation matters!



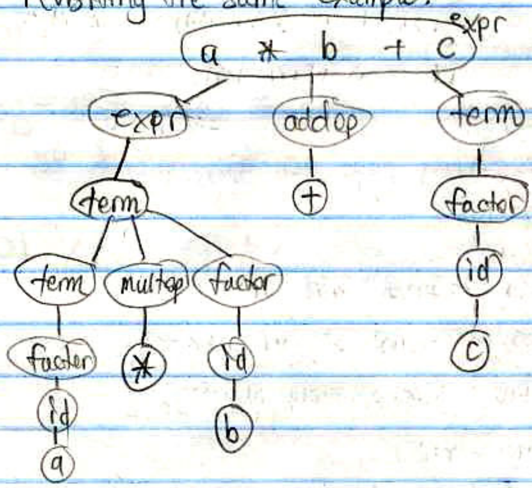
our grammer allows both!
— OR —



- we have ambiguity in our grammar → more than one possible parse tree.
- we must tell the computer what we really want to do by being precise.
- a couple ways we can fix this...
  - various compiler hacks can fix these things.
  - try and rewrite the grammar w/ less ambiguity. (code order of ops matter).

like this:
expr → term | expr addop term
term → factor | multop factor
factor → id | number | '(' expr ')'
addop → + | -
multop → * | /

revisiting the same example:



this time it takes into account the order of operation

→ in practice, context free grammar in full generally are a little too expensive to use in practice.

It turns out that only a subset is needed. People often use one of two parsable subsets... (we dont lose expressiveness)

Two flavors...

→ LL ⤳ Left to right, Leftmost derivation } we will end up using this for most of the course.

→ LR ⤳ Left-to-right, Rightmost derivation

LL PARSERS.

- relatively simple, can be constructed by hand
- top-down parsing, recursive decent
- also known as predictive parsers (sometimes look ahead, but still deterministic).
- a simple grammar for a list of args...

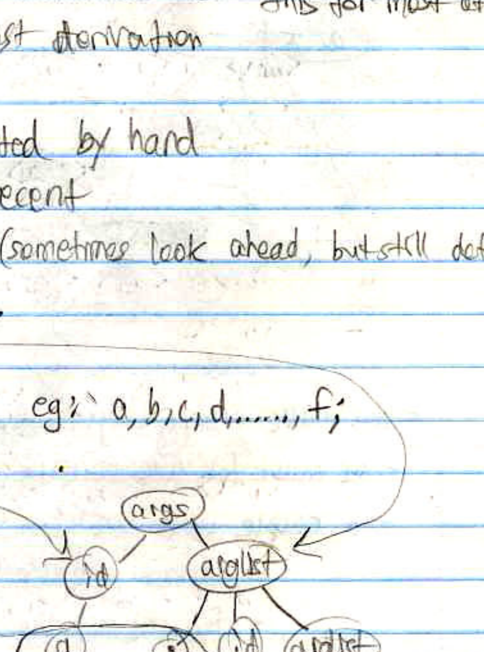
args → id arglist

arglist → ',' id arglist | ';'   eg: a, b, c, d, ..., f;

eg: input: a, b, c;

a , b , c ;

id



★ at each step we pick rule based on current token. ★

- sometimes we may not find a particular possibility. in which case we must scan forward one more token to see which rule it matches. This gives us another kind of grammar. If we look ahead n symbols, we have a LL(n) grammar. Most languages only need LL(1)

(note: try to minimize n)

COMP 302 SEPT 21 (CONT'D).

LR parsers → bottom up, parse tree is constructed during return trip of the recursion.

COMP 302 SEPT 23, 2009 PAUL HUSSMAN

Announcement: Office hours on Wednesdays 8:30am – 10:00am (½ hour earlier)

Last time: Parsers, CFGs (context free grammars) → In practice there are 2 simple languages → LL (predictive) (top down), LR (bottom up) (shift/reduce)

sometimes the first thing we see doesn't tell us exactly which rule to use so we have to look ahead up to n tokens this is called LL(n). In most cases, LL(1) is usually sufficient.

- consider the following grammar once again

  args → id arglist

  arglist → ',' id arglist | ';'

- an LR parser goes far into the tokens looking for ends of rules. In the following string: "a, b, c;", it sees 3 id's seperated by commas. once the semicolon is reached, it knows that everything before that is a rule title. It looks for which rule ends in a semicolon and then it knows that it's an arglist, Then it continues along and does the same thing. But first it checks if any other rules end in arglist. Two of them do, so it matches the longest rule. It ends up doing that 2 times (for each id in this example), and the first id has no comma, so its id arglist ⇒ args.
- Usually, LR parsers are written by tools (parser generators) ⇒ yacc, bison these will be table-based → complex, but efficient
- LL parsers are usually done by hand.
- As we parse, we can generate a datastructure, representing as everything we parsed.
  - → parse tree (Concrete Syntax Tree)
  - → Abstract syntax tree ⟵ no important difference for our purposes. (one's cleaner).
- Now we can process/analyze the code
- A simpler approach (historic approach) is a 1-pass approach
  - don't tend to build a data structure
  - Instead we output the translated/compiled code directly.
  - this impacted languages in a big way (historically).
  - In C for example (or Fortran), you have to specify all details since its 1-pass. this is a limitation.
  - Java is designed for multiple passes.

- Now we've seen parsing, now we want to do something with it.
  → Need a language to parse.
    - start off with a simple language, and add more complexity.
    - using the language 'wikitext'
      ↳ used in wiki's to express a simplified form of HTML for formatting text.
      - wikitext exists in many varieties. (not many standards).
      - one proposed standard: creole ← we will not use this.
      - we will use something closer to mediawiki's syntax (like wikipedia).
      - this language is not meant for programmers. Its philosophy is that most of the text that they write will show up into the wiki, and not have keywords. (naive text should not affect formatting).
        so the syntax can be 'ugly'.
    - Nb: we will not quite entirely follow mediawiki. (keep simple).

what does it actually look like?

the basic thing it should do:

- text should become text. If the user types lines of text, we have to figure out where the line breaks are, and we need to put <p> </p> markers around it.

- lists eg: (INPUT → CODE → HTML)

INPUT		CODE		HTML
* a	⇒	a	⇒	• a
* b		b		• b
* c		c		• c

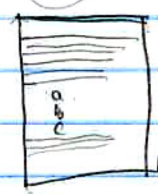
or

INPUT		CODE		HTML
# a	⇒	a	⇒	1 a
# b		b		2 b
# c		c		3 c

- Indenting

: a
: b
: c

definition list → <dl> <dd> a </dd>
<dd> b </dd>
</dl> <dd> c </dd> ⇒



tells browser to leave it alone and not to format it }

- preformatted blocks ␣abcd (↑ space) <pre> abcd </pre> ⇒ [abcd] ← unformatted.

PAUL HUSSMAN.

COMP 302 SEPT 23 (CONTD)

general approach

- read one line at a time (line parsing).
1) check if exists (EOF)
2) look at beginning and end of line. [B | ... | E]
   if B is a format code (*, :, #, space, etc) then process accordingly
3) if E is NOT a newline, result is just the input line.
4) Then it is a text paragraph (if 1-3 don't apply) <p>
   → paragraph will continue as text lines follow. Stop at blank line (just a NL) (new line)
   → what happens with multiple blank lines. ⟹ [NL NL NL NL abc]

So if a paragraph starts w/ a NL, then the text inside the paragraph will be "
"

Format Tokens

- always appear at the start of the line, otherwise considered just text,
- can be nested & mixed, arbitrarily. → *, #, :
  eg: ** a (unordered list with its first item as an unordered list w/ 'a' in it).

note: *a
*b
← special rules: blank line terminates list, but gets thrown away if present.
*c
*d

COMP 302 SEPT 25, 2009 PAUL HUSSMAN.

Last Time

→ LR parsing gets nasty by hand. We will use LL.
for doing LL, we can use functions like

```
function scan (str, tokens) {
}
function parse (str) {
    var t = scan(str, ...)
    switch (t.token) {
        case TOKEN10: ...
    }
```

} gets looped.
- call function to perform the rule action
- also need to consume / move past the token
* see textbook for template code. *

→ WML text processing

WML syntax → [Parser] → HTML syntax

- paragraph
- list ⟹ *, :, #, ␣ (always at beginning of line)
- list operators can be nested
- can also be mixed. eg; *:*#

- main idea for parsing

*# abc
*# def } new item of existing
*##} new sublist
*} back down to no nestings.

} we can get an algorithm for this.

Hilroy

PAUL HUSSMAN.

→ <u>how does list nesting change?</u>

→ current list level (char/token)
→ previous list level (char/token)

— if the current level adds to the previous list format, then we need to open up a new list, corresponding to what we actually have.
→ for each * we need , for each # we need , ... etc

— if current & previous are the same
→ close the previous & add a new item: , or </dd><dd>

— if the string is smaller by N lists: () N times.

— if our list format is not empty (after we cut of N chars): ...

eg: *a → a
*b → b
→

→ demo ~~for assignment~~ → how this stuff works, example.

→ <u>what are we doing?</u>

WE ARE DOING <u>1-PASS CODE-GENERATION</u>.

- what else might we want to generate?
→ the part we haven't thought about yet... <u>CONTROL FLOW</u>.

<u>CONTROL FLOW</u>
— most compilers use a template approach to (initial) code-generation.
— what about expressions?
[a + b] ⟶ load r1 with a, load r2 with b, add r_1, <>, r_1

→ this leads to complexity
- some operators are [overloaded] (different meanings in different contexts)
eg: '+' means <u>add</u> for numbers and <u>concatenate</u> for strings.

- [order of operations]
eg: a + b * c
- for simple cases, this can be taken care of with the grammar.
- we can also add <u>EXPLICIT PRECEDENCE</u> (a heirarchy → how tightly things group).
eg: '*' has higher precedence than '+', ∴ '*' is done first.
- classic example: language PASCAL doesn't have precedence. eg: a<b & c<d

- [associativity]
eg: 10 − 3 − 2 = 5 left associativity ✓
10 − (3 − 2) = 9
- a left associative operator groups to the left.
- a right associative operator groups to the right. [eg: exponentiation]

- [assignments]
eg: a = b + c   right of 'equals' sign.
a = 2;
x = a; // a here is an R value (we think of its contents)
a = y + z; // a here is an L value (we think of its address).
- a variable is a name container
↳ has <u>address</u> & <u>contents</u>
- we could also think of variables as references

the value model { x = 2   x[2]
y = x   y[2]
z = x + y   z[4]

the reference model (less common) { x → 2 // x REFERS to 2
y ↗
z → 4 // once computation has occurred

COMP 302 SEPT 28, 2009

PAUL HUSSMAN.

Last time → associativity, assignments w/ context (etc).

- [coercion] → need to make sure the types match

  int a;
  double d;
  d + a ⇒ ?

  doubles hold integers, so stuff gets converted to doubles.

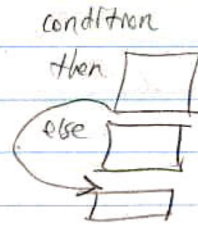
- recall value vs reference model...
  ↳ named containers ↳ named references

- evaluation order
  eg: a + b
  - evaluate a, evaluate b, add the values.
  - we chose to evaluate left to right (a first, then b).
  - we could do it the other way
  - this applies to operators and function calls.
  - in many/most cases the order doesn't matter, but there are times when it matters.
    it matters if the arguement has a side-effect
    ie change the global state of data (eg: I/O)
    eg: (x=read();) + (x+1) ⇐ ORDER MATTERS.
  - so this stuff needs to be defined. (like in Java)
  - is x++ the same as x=x+1 ?
    in Java & C, they are the same ⇒ x++ is syntactic sugar for x=x+1
    in Javascript, eg: x = "0"
    x++ → 1
    x=x+1 → "01"

CONDITIONALS → every language needs ways of choosing things.

if-statement → condition, then, else ⇒ condition → then / else → result

evaluate condition test, branch



→ sometimes people don't actually evaluate conditionals
→ boolean short-circuit in conditionals

a && b — eval a, eval b, end — evaluate a → test, branch, evaluate b → test, branch

→ switch-statement
→ basically a multiway if statement.

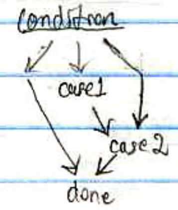
condition → case1, case2, case3, ..., default → done

- from the conditional, it is figured out which case is used.
- in Pascal/C, it can be thought of as branching

condition → case 1 → case 2 → ...

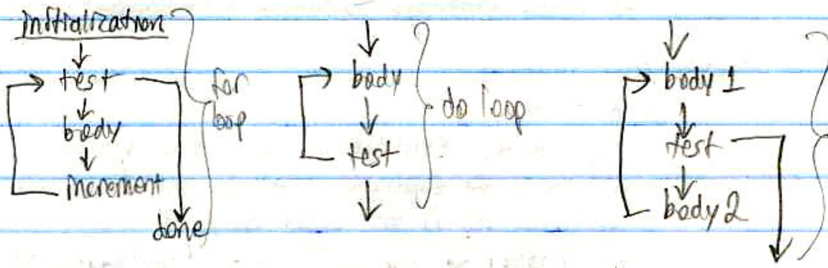
- in Javascript it is a series of if-statements
  - the cases are tested in sequence. Hilroy
  → fall-through paths ⇒ case, break

- with a fallthrough feature...
  - it gets conceptually more complex
    - depends what you want to present to the user.



LOOPS

- explicit loops → different flavours → for, while, do.



mid-loop exits
uses break & continue

FUNCTIONS

- programs need control abstraction → extracting chunks of code

consider: foo(a,b) — actual arguements
vs, — formal parameters
function foo (a,b) { ... }

→ when functions are called, certain things happen →
- evaluate in order
- store the results of the evaluation
  - → stacks
  - → registers
- store the return address
  - → stack
- jump to procedure code
- return

→ what form do parameters
- → call by value
- → call by reference

```
function foo (a,b) {
   a=1
}  b=2
c=2; d=3;
foo (c,d)
```
→ c,d = ?

- notice we pass the value to the arguement, not the variable itself.

- in call by reference

foo (a,b) {
↕ ↕
foo (c,d)

c and a are identifiers
if a is changed, c is changed
b, d are identifiers.

- in our example → we should get 1, 2 with call by reference.

C, Java, JS → call by value

C++ has both, but mostly call by value

```
swap (int &a, int &b) {   ← a and b are reference parameters
   int t = a
   a = b
}  b = t
```

PAUL HUSSMAN

COMP 302 SEPT 28 (CONTD)
SEPT 30

x=1, y=2
swap (x,y)
x==2, y==1

SEPT 30, 2009

- Last time → control flow, expressions, control flow, sequences of statements, conditionals, loops, recursion instead of iteration (more later), functions/procedures/methods/subroutines,
  → ways of passing parameters → call by value (C & Java), call by reference → the same variable might be referenced by two names.
  → call-by-value sort of does shortcuts
    example in large data structures (if you wanna pass a large array.
      foo (a)
      foo (b) {,,, }
      if a large array is passed, then copying everything is EXPENSIVE.
    → in languages that use call by value, large data structures often have special case → call by reference.
  → call by sharing → variation
    → pointer model
    → foo(a){ ⇒ a lies in memory and points to other memory.
      } foo (b) ⇒ points to same memory.

- DEFAULT PARAMATERS
  can you call a function without all of the arguements?
  if I define foo(a,b,c) can I call foo(1,2)? [Not in Java, but yes in JS]
  Javascript will assume 'undefined' for c.

- NAMED PARAMETERS
  so far we have considered POSITIONAL PARAMETERS
  function defined as foo(a,b) {...}
  when called: foo(c, d)
  OR we could make it explicit based on an assignment instead of making it implicit based on order,
  - in this case when called it will look like foo (a=c, b=d)
  - also the name is flexible if not all args are given.
    - with positional, its hard to skip a parameter. Something must be specified.
    - with named we can just skip b like this: foo (a=1, c=2, d=3)

- So far: EAGER EVALUATION:
  before we can do any operations, we need to evaluate the arguements first!
  eg: foo(a, b) ⇒ First evaluate a & b, then apply code in foo().
  - this is not always optimal
  eg: function foo (a,b) {
        if (a>0)
          return 8
        return foo(b,a)
      }
  } there is a code path here where we do not need to know what b is.
  what if I tried to call foo(1, foo(0,0));
  with eager evaluation this will be an infinite loop.

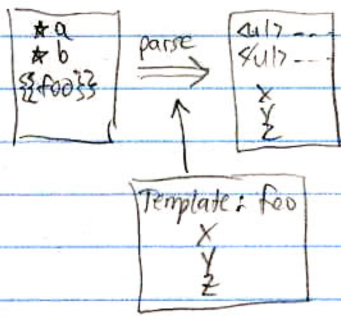
- other Method: LAZY EVALUATION
  - evaluate what you need, when you need it.
  - above code will return 8. ⇒ avoids the infinite loop too.

Hilroy

- LAZY VS EAGER EVALUATION
  - In what context do we evaluate
    - eg: g = 1 //glbl
      foo(∅, g+1); ⇒ g would be 2
      ⋮
      g = 2
      if (b ....){ ⇒ g would be 3

      which one do we actually want?
      - we evaluate an environment
        → mapping from args to current value
      - we will usually want the original environment from the time that it was called.
  - In lazy evaluation we want to package the environment and expression together
    this is called a **THUNK** ≡ Environment and an expression
    ↳ args to values;

- In our WML language...
  → currently we have no control flow.
  → we will use templates in wikis to create control abstraction
  → we can use Macro Expansions to say: this chunk of text should go everywhere.
  → make a new webpage, write code, and refer to it from the wiki.



  → so we can start to think of templates as function calls.
  → in fact, templates do allow parameters.

    {{foo | a | b | c}}
    → template call
    → pipe
    → parameter

  → we can put anything we want as parameters
  → arbitrary white space within template call is thrown away.
  → how do we do stuff?
    - referring to parameters → simplest way is positional ⇒ uses 3 curley braces.
      continuing above example, {{{1}}} represents the FIRST parameter. (this goes in template code)
    - it also accounts for pass by name
      {{ foo | a=1 | b=2 | c }} ⇒ so {{{a}}} gives a 1
      (a=1, b=2: named; c: positional) {{{b}}} gives a 2
      {{{1}}} gives a 1
    - Positional parameters are counted as if named parameters were not there. They are counted independently of any interweaving named parameters.
    - what is the value of {{{abc}}} if 'abc' is not actually passed?
      → wikis just consider it text (if it is not an existing parameter) "{{{abc}}}"
      → {{{abc|}}} → if abc is defined then fine → go normal
        otherwise use stuff on the right of the verdical bar → in this case its an empty string

COMP 302 OCT 2, 2009 PAUL HUSSMAN.

Last Time: finished up code generation, control flow, functions

- how can we incorporate control abstraction into WML. ⇒ Templates
- default value for params → if you refer to {{{bar}}}, but it is not supplied as a parameter, then it just becomes text ⇒ "{{{bar}}}", OR you can say {{{bar|}}} will say use param bar if it is defined, otherwise it evaluates to whatever is on the right of the pipe symbol. In this case it is an empty string.
- for us, templates will have to be in the same page.
- we will need some way of making choices & some way of iterating (recursion).
- we will need a special syntax to define templates in order to embed them into our same page.

  {: foo | a b c | body :}

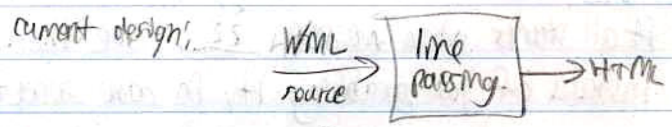
  name (pipe) list of params, body of the template

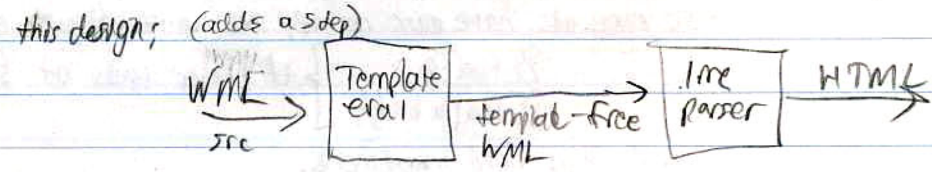
- template invocation

  {{foo | arglist }}

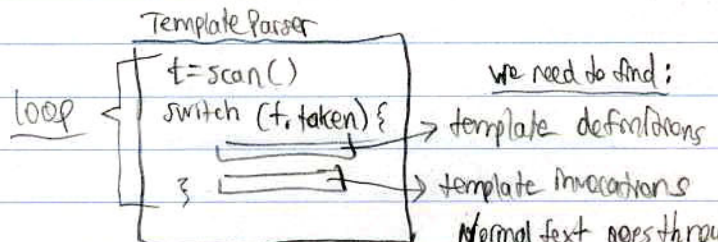
(makes it more complicated) - in our parameter list, in our template names, in our args... etc, we will have to allow for recursive template invocations & definitions!

- we will have to add a template evaluating pass to your system.





- what does template eval do?
  - we evaluate text as our input. (parsing)
  - when we encounter template invocations, we actually have to do something.



- Evaluate definition
  - once we see this symbol ⇒ {:, we need to recursively investigate the text inside.
  - we expect the template name to come first. ⇒ names cannot have '|', ':', '{', '}'
  - how do we know when we stop? → at a '|' symbol.

- If we do not stop at a '|', we will stop at ":}" (just to be safe).
- after the name comes a list of arguements.
  - these are simpler identifiers → no spaces
  - we need to extract the list of arguements (stops at a '|') ↙ or ":}"
- then comes the template body itself.
  - this is everything after, until the :}
  - we need to extract the body. In most languages when we parse for definitions, we do NOT actually execute the function body. (we wait until it is called).
  - But what if there is a nested feature within the body?
    ∴ we DO need to parse through the body to look for nested things.
  then we need to find the matching :}.
- Now we have everything that we need.
  ⇒ name, arglist, body
  - now we are ready to consider calling/invoking this function.
  - we must keep all the template definitions in a structure, so that we can easily find them by name.

Invocation

- It all starts when we see {{. we then must evaluate the name, instead of just grabbing it, in case something is nested within it.
- so once we have our name, we must figure out where to end it.

  {{foo}}
  {{foo|a b c}} } it either ends at }}, or a '|'.

- next step: need arguements.
  → recursively evaluate until we get to another '|', or }} (keep doing this while '|' still exists)

  {{foo}}
  {{foo|a}}
  {{foo|a|b}}

COMP 302 OCT 5, 2009 (from Meki) PAUL HUSSMAN

Last Time: - adding control abstraction to WML
- template (macro replacement)
- think of it as functions
ex: {{foo}} template
- want to mix recursively
- repeat until you need to stop ⇒ run out of string, or hit a stopping token
- at each point when we reach the next token,
- call evaluate (recursively)
- eval keeps going unless it hits a token that stops recursive parse
{{name}} - we look for this recursively, until we get a }} or |

name based, position based parsers
ex: {: hello| you| Hi <b {{{you}}} :}
- recognize {: (definition)
- look for more
1st recursive call - start at a '|'
- error handling → could also stop at :}
2nd recursive call - {: eval hello → evaluate that template
- body → not recursive
- need to find the eval. despite only a string, but do not interpret it.

{{hello| dark}}
{{
name ⇒ hello. eval... | or }}
get params if any
evaluate |}}
- once we evaluate the complete invocation,
find function definition from the name
name: {arglist: ~~
body: ~~}
- next we stop → associate or bind our actual arg values to the formal params.

var → value
•

list of bindings
environment
- execute: (ie evaluate expression recursively) [body in this context]

another approach; substitution
- replace all instances of formal arguements in our function body with the actual values.

Hi {{{you}}}

→ first as in invocation → clark
    associate you : clark
    look for {{{you}}} or {{{you|}}}

  - we can do this as a simple text replacement.
    {{{ {{{ x }}} }}}
        ← →

  - in practice, parsers must not move nested invocations, definitions, etc.

Result of substitution: Clark

  - now could just return done
    but INSTEAD we will (RECURSIVELY) evaluate the body
    (here, there are no nested invocations ___)

This gives us functional abstraction, but we still don't have conditionals

- {{{ a| }}} means (IF) a is defined, then a, (ELSE) whatever is on right of '|'.

- arithmetic
  - could build it into the grammar (but we won't here)
  - we will side step this issue
  - we will introduce some internal function.

COMP 302 OCT 7, 2009 PAUL HUSSMAN

Last Time — template system → parse(eval) def^n s, invocations
notes trick for definitions → don't evaluate body.

- invocation
  steps: - parse eval
  - lookup
  - arg match
  - body expansion
- we still don't have - recursive evaluation
  arithmetic/conditional ability
  - we do have a flavour of conditionals: {{{arg|}}}
    if arg is defined, then use it, otherwise/else what's on right of '|'

ARITHMETIC

- we could call & develop a whole grammar of expressions, but this is a lot of work
- instead we use the existing template mechanism to open up a hole between the WML & the javascript.
- special functions allow us to do this
  all special functions start with a #
  n.b. in WIKI TEXT, its slightly different.

common syntax wiki syntax
{{#expr|...}} {{#expr:...}}

- we will use the COMMON SYNTAX.
- it will be predefined in JAVASCRIPT source code.
  - takes 1 (positional) arguement.
    - takes an arbitrary arithmetic expression which gets passed to the JavaScript eval function.
      eval("3+5*2") → 13
      ↳ string of arbitrary JavaScript Code
      NOTE: this is a BIG security hole!
      → be careful of crashing the browser
      → to be more secure, we need to inspect & verify that it is a regular old arithmetic expression
      → we can allow things such as Math.floor, Math.abs, etc...

CONDITIONALS

- we can add conditionals in a similar way as arithmetic
  - true is any non-empty string
  - false is an empty string.

{{#if | conditional | true part | false part}} ← function that takes 3 arguements & returns the eval() result of one of the last two.

NORMALLY,

if (condition)
    true part
else
    false part

we evaluate the condition, then CHOOSE which part to evaluate based on true or false.

FOR THIS CASE

{{ #if | condition | true part | false part }}

- here we would normally evaluate each arguement, regardless of the condition
- we must first evaluate the condition, and then discard one arguement.
- the #if functions must be parsed differently!

EXAMPLE

{: foo | a | {{ #if | {{{a|}}} | {{ foo | a }} | ok }} :}

{{ foo }} → ok

{{ foo | x }} → infinite loop

If we evaluate all the args in the if-template, eagerly/right away → recursion

ADDITIONALLY

There are a few other flavors of the #if

- #ifeq (if equal)

  {{ #ifeq | a | b | truepart | falsepart }}

  (if a == b, then true part else false part)

- #switch (nested if), also exists.

regular logic: if start ≥ stop return start, else (recursively iterate, increment, stop)

EXAMPLE

{: count | start stop | {{#ifeq | {{ #expr | {{{ start }}} ≥ {{{ stop }}} }} | true | {{{ start }}} | {{{ start }}} {{ count | {{ #expr | {{{ start }}} + 1 }} | {{{ stop }}} }} }} :}

SCOPES / BINDINGS

{: foo | ... | ...

foo → { arglist : ___
        body : ___ }

→ when we define templates, we are creating bindings.

(binding is association of two things)

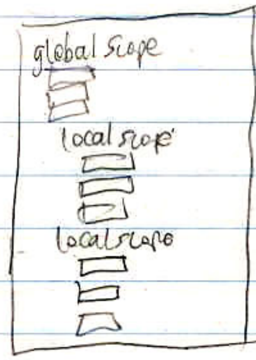
→ we need to think about when vars & there values are bound, and for how long, and who/what can actually see them, (visibility). (So far we are in one big global environment).

COMP 302 OCT 7 (CONT'D). PAUL HUSSMAN

binding created
⋮
binding dies
} lifetime

} often, lifespan lies within the SCOPE of the language.
Eg: a method variable is only bound within the method, then it dies.

- STATIC scoping (most common), DYNAMIC scoping (rare)
  - also known as lexical scoping
  - lifetime of a variable is defined textually. (curley braces).
  - this is straight forward
  - for this to work, we need multiple, NESTED scopes.
    global scope contains all other (nested) scopes. i.e. local scopes
  - allows for multiple layers of nesting
    ∴ we have arbitrary scopes within scopes



  - local scopes can arise from a function definition.
    - we can nest function definitions (like MJS),
    - or we can use compound statements in C & Java. ⇒ {...}

NOTE - Javascript has a slightly different model of visibility in scope
  - if we have the same variable in a nested scope
  - everything is flattened.

```
var x=1;
  {var x=1;
     {var x=2;
     }
  } return x
```