Lecture 11

Implementing Small Languages

internal vs. external DSLs, hybrid small DSLs

Ras Bodik
Shaon Barman
Thibaud Hottelier

Hack Your Language!

CS164: Introduction to Programming Languages and Compilers, Spring 2012

UC Berkeley

Where are we?

Lectures 10-12 are exploring small languages

- both design and implementation

Lecture 10: regular expressions

- we'll finish one last segment today

Lecture 11: implementation strategies

- how to embed a language into a host language

Lecture 12: problems solvable with small languages

- ideas for your final project (start thinking about it)

Today

Semantic differences between regexes and Res

Internal DSLs

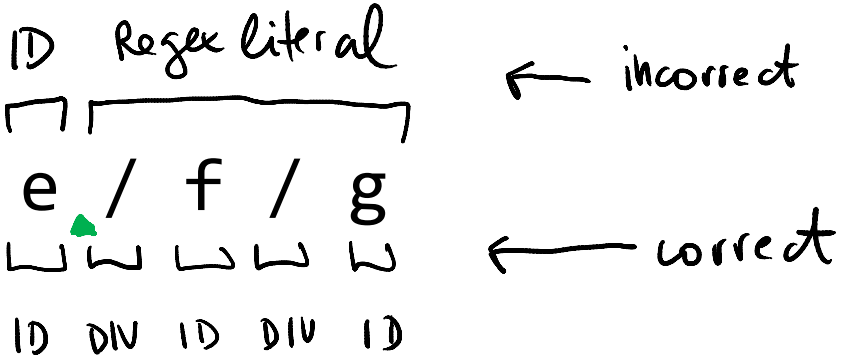
Hybrid DSLs

External DSLs

Answer to 2nd challenge question from L10

Q: Give a JavaScript scenario where tokenizing depends on the context of the parser. That is, lexer cannot tokenize the input entirely prior to parsing.

A: In this code fragment, / / could be div's or a regex:



parser needs to tell lexer that at ▲, an op is expected, not an expression

Recall from L10: regexes vs REs

Regexes are implemented with backtracking

- This regex requires exponential time to discover that it does not match the input string X==========.

- regex: X(.+)+X

REs are implemented by translation to NFA

- NFA may be translated to DFA.
- Resulting DFA requires linear time, ie reads each char once

The String Match Problem

Consider the problem of detecting whether a pattern (regex or RE) matches an (entire) string

match(string, pattern) --> yes/no

The regex and RE interpretations of any pattern agree on this problem.

That is, both give same answer to this Boolean question

Example: X(.+)+X

It does not matter whether this regex matches the string X===X with X(.)(..)X or with X(.)(.)(.)X, assigning different values to the '+' in the regex. While there are many possible matches, all we are about is whether *any* match exists.

Let's now focus on when regex and RE differ

Can you think of a question that where they give a different answer?

Answer: find a substring

Example from Jeff Friedl's book

Imagine you want to parse a config file:

```
filesToCompile=a.cpp b.cpp
```

The regex for this command line format:

```
[a-zA-Z]+=.*
```

Now let's allow an optional \n-separated 2nd line:

```
filesToCompile=a.cpp b.cpp \<\n>
               d.cpp e.h
```

We extend the original regex correspondingly:

```
[a-zA-Z]+=.*(\\\n.*)?
```

This regex does not match our two-line input. Why?

What compiler textbooks don't teach you

The textbook *string matching* problem is simple:

*Does a regex r match the **entire** string s?*

- a clean statement suitable for theoretical study
- here is where regexes and FSMs are equivalent

In real life, we face the *sub-string matching* problem:

*Given a string s and a regex r, find a **substring** in s matching r.*

- tokenization is a series of substring matching problems

Substring matching: careful about semantics

Do you see the language design issues?

- There may be many matching substrings.
- We need to decide **which** substring to return.

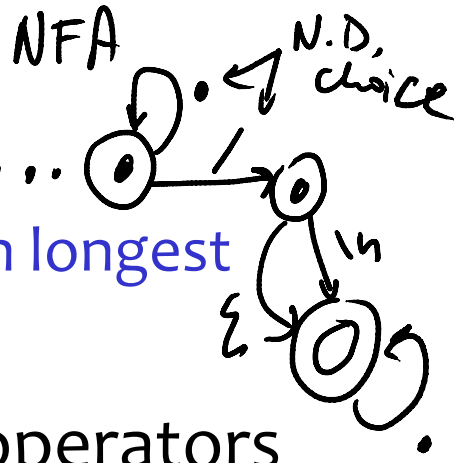
It is easy to agree where the substring should **start**:

- the matched substring should be the **leftmost** match

They differ in where the string should **end**:

- there are two schools: RE and regex (see next slide)

Where should the matched string end?



Declarative approach: longest of all matches

- conceptually, enumerate all matches and return longest

Operational approach: define behavior of *, | operators

e* match e as many times as possible while allowing the remainder of the regex t o match (greedy semantics)

e*|*e select leftmost choice while allowing remainder to match

```
filesToCompile=a.cpp b.cpp \<\n> d.cpp e.h
```

```
[a-zA-Z]+  = .* ( \\ \n .* )?
```

These are important differences

We saw a non-contrived regex can behave differently

- personal story: I spent 3 hours debugging a similar regex
- despite reading the manual carefully

The (greedy) operational semantics of *

- does not guarantee longest match (in case you need it)
- forces the programmer to reason about backtracking

It may seem that backtracking is nice to reason about

- because it's local: no need to consider the entire regex
- cognitive load is actually higher, as it breaks composition

Where in history of *re* did things go wrong?

It's tempting to blame perl

- but the greedy regex semantics seems older
- there are other reasons why backtracking is used

Hypothesis 1:creators of re libs knew not that NFA can

- can be the target language for compiling regexes
- find all matches simultaneously (no backtracking)
- be implemented efficiently (convert NFA to DFA)

Hypothesis 2: their hands were tied

- Ken Thompson's algorithm for re-to-NFA was patented

With backtracking came the greedy semantics

- longest match would be expensive (must try all matches)
- so semantics was defined greedily, and non-compositionally

Regular Expressions Concepts

- Syntax tree-directed translation (re to NFA)
- recognizers: tell strings apart
- NFA, DFA, regular expressions = equally powerful
- but \1 (backreference) makes regexes more pwrful
- Syntax sugar: e+ to e.e*
- Compositionality: be weary of greedy semantics
- Metacharacters: characters with special meaning

Internal Small Languages

a.k.a. internal DSLs

Embed your DSL into a host language

The host language is an interpreter of the DSL

Three levels of embedding

where we draw lines is fuzzy (one's lib is your framework)

1) Library
2) Framework (parameterized library)
3) Language

DSL as a library

When DSL is implemented as a library, we often don't think of it as a language

- even though it defines own abstractions and operations

Example: network sockets

```
Socket f = new Socket(mode)
f.connect(ipAddress)
f.write(buffer)
f.close()
```

GUI

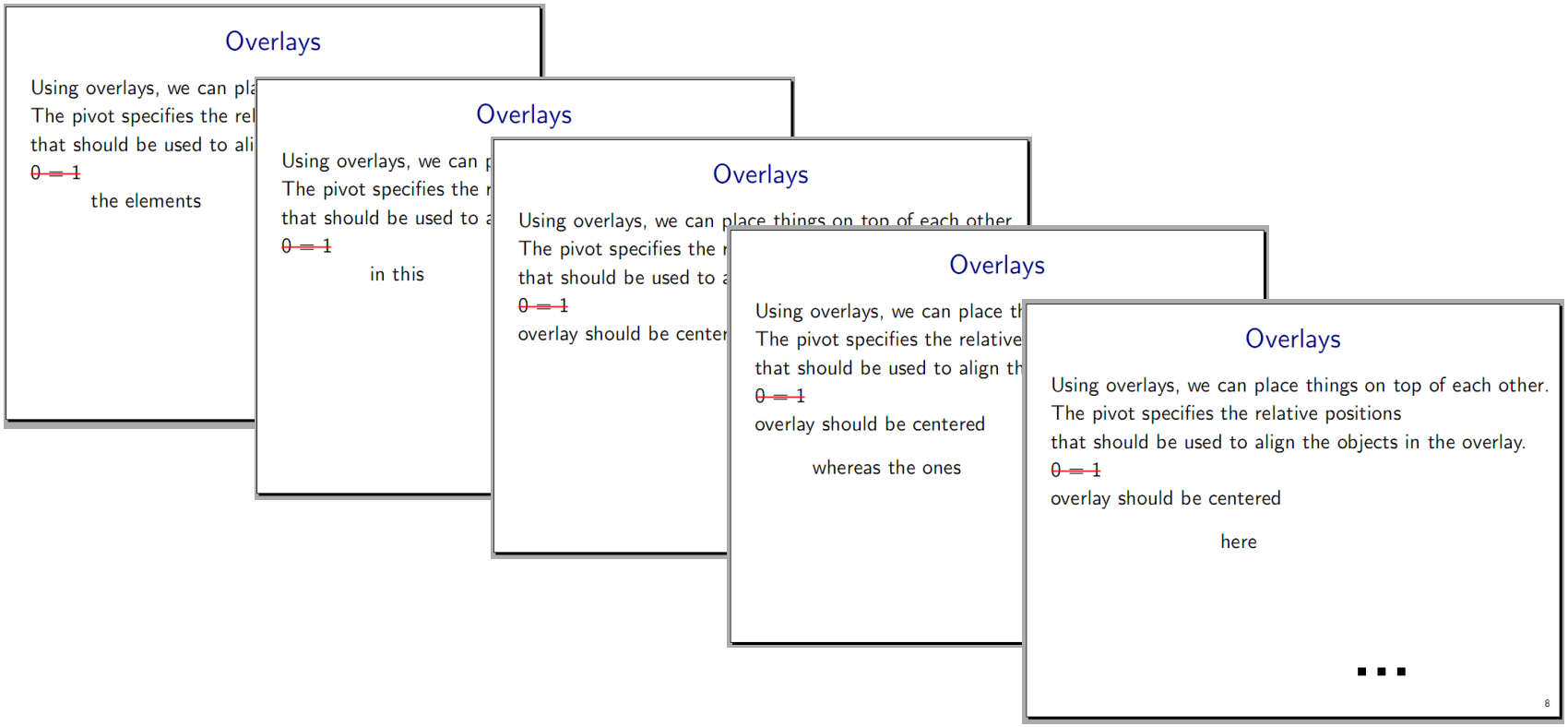
Gem Ruby

rspec

The library implementation goes very far

rfig: formatting DSL embedding into Ruby.

see slide 8 in http://cs164fa09.pbworks.com/f/01-rfig-tutorial.pdf



The animation in rfig, a Ruby-based language

```
slide!('Overlays',

  'Using overlays, we can place things on top of each other.',
  'The pivot specifies the relative positions',
  'that should be used to align the objects in the overlay.',

  overlay('0 = 1', hedge.color(red).thickness(2)).pivot(0, 0),

  staggeredOverlay(true, # True means that old objects disappear
    'the elements', 'in this', 'overlay should be centered', nil).pivot(0, 0),

  cr, pause,            # pivot(x, y): -1 = left, 0 = center, +1 = right

  staggeredOverlay(true,
    'whereas the ones', 'here', 'should be right justified', nil).pivot(1, 0),
  nil) { |slide| slide.label('overlay').signature(8) }
```

DSL as a framework

It may be impossible to hide plumbing in a procedure

these are limits to procedural abstraction

Framework, a library parameterized with client code

- typically, you register a function with the library
- library calls this client callback function at a suitable point
- ex: an action to perform when a user clicks on DOM node

Example DSL: jQuery

Before jQuery

```
var nodes = document.getElementsByTagName('a');
for (var i = 0; i < nodes.length; i++) {
    var a = nodes[i];
    a.addEventListener('mouseover', function(event) { event.target.style.backgroundColor='orange'; }, false );
    a.addEventListener('mouseout', function(event) { event.target.style.backgroundColor='white'; }, false );
}
```

jQuery abstracts iteration and events

if nodes n s.t. n is an a node ...

```
jQuery('a').hover( function() { jQuery(this).css('background-color', 'orange'); },
                   function() { jQuery(this).css('background-color', 'white'); } );
```

Embedding DSL as a language

Hard to say where a framework becomes a language

not too important to define the boundary precisely

Rules I propose: it's a language if

1) its abstractions include compile- or run-time checks --- prevents incorrect DSL programs

ex: write into a closed socket causes an error

2) we use syntax of host language to create (an illusion) of a dedicated syntax

ex: jQuery uses call chaining to pretend it modifes a single object:

```
jQuery('a').hover( … ).css( …)
```

rake

rake: an internal DSL, embedded in Ruby

Author: Jim Weirich

functionality similar to make

- has nice extensions, and flexibility, since it's embedded
- ie can use any ruby commands

even the syntax is close (perhaps better):

- embedded in Ruby, so all syntax is legal Ruby

http://martinfowler.com/articles/rake.html

Example rake file

```
task :codeGen do
  # do the code generation
end

task :compile => :codeGen do
  # do the compilation
end

task :dataLoad => :codeGen do
  # load the test data
end

task :test => [:compile, :dataLoad] do
  # run the tests
end
```

Ruby syntax rules

Ruby procedure call

foo 1 2 lamdda

dict. literal

:a => :b

in Python: {"a": "b"}

↑ key ↑ value

How is rake legal ruby?

Deconstructing rake (teaches us a lot about Ruby):

```
task :dataLoad => :codeGen do
  # load the test data
end

task :test => [:compile, :dataLoad] do
  # run the tests
end
```

Two kinds of rake tasks

File task: dependences between files (as in make)

```
file 'build/dev/rake.html' => 'dev/rake.xml' do |t|
  require 'paper'
  maker = PaperMaker.new t.prerequisites[0], t.name
  maker.run
end
```

Two kinds of tasks

Rake *task*: dependences between jobs

```
task :build_refact => [:clean] do
  target = SITE_DIR + 'refact/'
  mkdir_p target, QUIET
  require 'refactoringHome'
  OutputCapturer.new.run {run_refactoring}
end
```

Rake can orthogonalize dependences and rules

```
task :second do
  #second's body
end

task :first do
  #first's body
end

task :second => :first
```

General rules

Sort of like make's %.c : %.o

```
BLIKI = build('bliki/index.html')

FileList['bliki/*.xml'].each do |src|
  file BLIKI => src
end

file BLIKI do
  #code to build the bliki
end
```

Parsing involved: DSL in a GP language

GP: general purpose language

JS: `if ("___".match(/___/))`

DSL, in JS

C preprocessor:

templating

jQ has a selector language

LINQ in C#

Parsing involved: GP in a DSL language

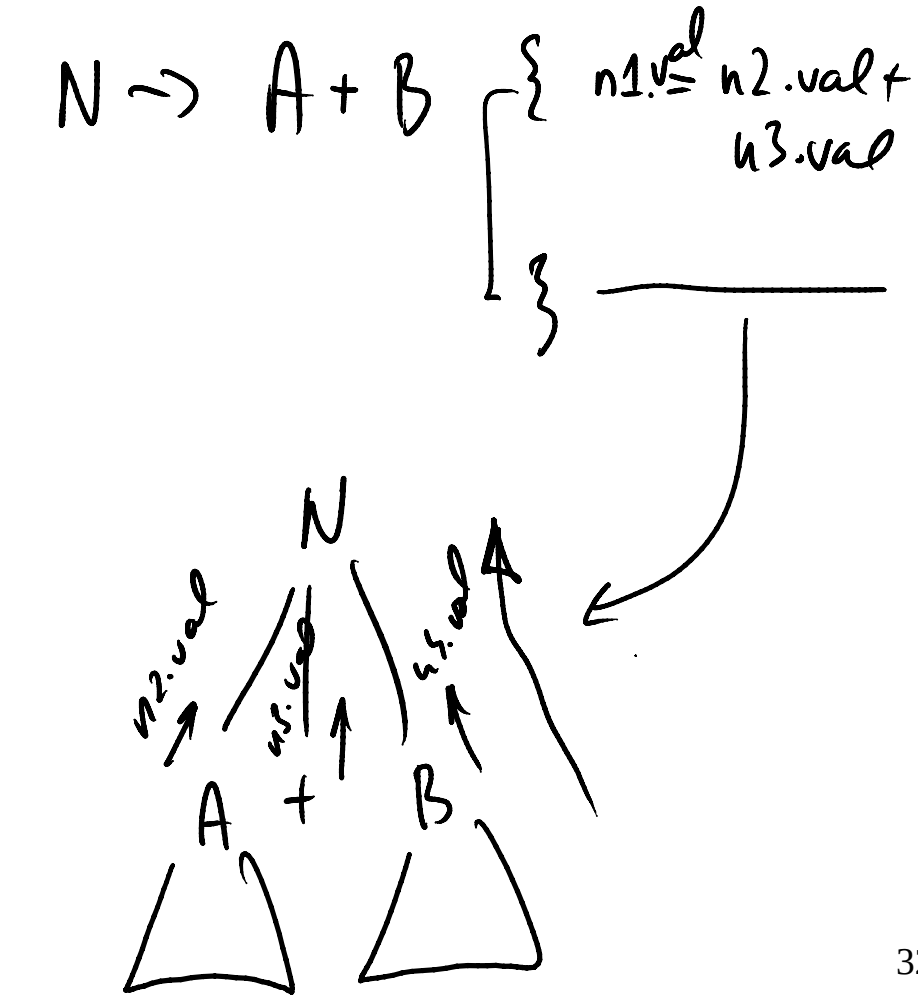
GP: general purpose language

lua in TeX

shader language

JS in HTML

perl in Regex



External DSL

Own parser, own interpreter or compiler

Examples we have seen:

Reading

Read the article about the rake DSL

Acknowledgements

This lecture is based in part on

Martin Fowler, “Using the Rake Build Language”

Jeff Friedl, “Mastering Regular Expressions”