HLT

Introduction to Xerox Finite State Tool (xfst)

University of Malta

Acknowledgements

- Shuly Wintner, Lecture Notes, 2008

Outline

Outline

Outline

Outline

Outline

Outline

Outline

What is `xfst`

- `xfst` is an interface giving access to finite-state operations (algorithms such as union, concatenation, iteration, intersection, composition etc.)
- `xfst` includes a regular expression compiler
- `xfst` is bidirectional. The interface includes
  - a lookup operation (apply up)
  - a generation operation (apply down)
- The regular expression language employed by `xfst` is an extended version of standard regular expressions

Atomic Expressions

- `0` the epsilon symbol denotes the empty string language
- `?` the any symbol deotes the language of all single-symbol strings or the corresponding identity relation. The empty string is not included.
- Any single symbol `a` denotes the language consisting of the corresponding string or the identity relation on that language
- Any pair of symbols `a:b` denotes the relation that consists of the corresponding ordered pair of strings. `a` is the *upper* symbol and `b` is the *lower* symbol. A pair of identical symbols, except for the pair `?:?` is considered to be equivalent to the corresponding single symbol
- What is the relationship between `?` and `?:?`
- `cat` a single *multicharacter symbol*
- `"+Noun"` single symbol with multicharacter print name
- `%+Noun` single symbol with multicharacter print name

Atomic Expressions

`cat`	a single *multicharacter symbol*
`%+`	the literal plus-sign symbol
`%*`	the literal asterisk symbol (and similarly for `%?`, `%(`, `%]` etc.)
`"+Noun"`	single symbol with multicharacter print name
`%+Noun`	single symbol with multicharacter print name

Operators

`[ ]`	empty string
`[A]`	same as `A`
`A | B`	union
`(A)`	optional A
`A & B`	intersection
`A-B`	set difference

Symbols and Operators

`A B`	concatenation
`c a t`	language consisting of the string "cat"
`{cat}`	language consisting of the string "cat"
`A*`	Kleene Star (zero or more iterations)
`A+`	one or more iterations
`?*`	the universal language
`~A`	complement of a (= `[?* - A]`)
`~[?*]`	the empty language

- Question: What is the difference between `~[?*]` and 0

Symbols and Operators

`A B`	concatenation
`c a t`	language consisting of the string "cat"
`{cat}`	language consisting of the string "cat"
`A*`	Kleene Star (zero or more iterations)
`A+`	one or more iterations
`?*`	the universal language
`~A`	complement of a (= `[?* - A]`)
`~[?*]`	the empty language

- Question: What is the difference between `~[?*]` and 0

Relations

`[A .x. B]`	Cartesian product relating every string in A to every string in B
`a:b`	same as `[a .x. b]`
`%+PLU:s`	same as `[+PLU .x. s]`

Abbreviations

- `$A`: all strings that contain `A`
  - Question: How would you define `$A` using concatenation?
  - Answer `$A = [?* A ?*]`
- `A/B`: the language obtained by splicing in B* anywhere within the strings of A strings from B
  - example `[[a b] / x]`: includes `xxxaxxbxx`
- `\A`: The set of all single symbol strings tat are not in the language A.
  - example `\b: [? - b]` - any symbol except B

Abbreviations

- `$A`: all strings that contain `A`
  - Question: How would you define `$A` using concatenation?
  - Answer `$A = [?* A ?*]`
- `A/B`: the language obtained by splicing in B* anywhere within the strings of A strings from B
  - example `[[a b] / x]`: includes `xxxaxxbxx`
- `\A`: The set of all single symbol strings tat are not in the language A.
  - example `\b`: `[? - b]` - any symbol except B

Abbreviations

- `$A`: all strings that contain `A`
  - Question: How would you define `$A` using concatenation?
  - Answer `$A = [?* A ?*]`
- `A/B`: the language obtained by splicing in B* anywhere within the strings of A strings from B
  - example `[[a b] / x]`: includes `xxxaxxbxx`
- `\A`: The set of all single symbol strings tat are not in the language A.
  - example `\b`: `[? - b]` - any symbol except B

User Interface

```
xfst> help
xfst> help union net
xfst> exit
xfst> read regex [d o g | c a t];
xfst> read regex < myfile.regex
xfst> apply up dog
xfst> apply down dog
xfst> pop stack
xfst> clear stack
xfst> save stack myfile.fsm
```

User Interface

```
xfst> define Root [w a l k | t a l k | w o r k];
xfst> define Prefix [0 | r e];
xfst> define Suffix [0 | s | e d | i n g];
xfst> read regex Prefix Root Suffix;
xfst> words
xfst> apply up walking
```

Replace Rules

A Motivating Example

Consider the following pairs:

Example

accurate inaccurate	adequate inadequate	balanced imbalanced	competent incompetent
definite indefinite	finite infinite	mature immature	nutrition innutrition
patience impatience	possible impossible	sane insane	tractable intractable

- The negative forms are constructed by adding the abstract morpheme iN to the positive forms.
- N is realized as either n or m.

Consider the following pairs:

Example

accurate inaccurate	adequate inadequate	balanced imbalanced	competent incompetent
definite indefinite	finite infinite	mature immature	nutrition innutrition
patience impatience	possible impossible	sane insane	tractable intractable

- The negative forms are constructed by adding the abstract morpheme **iN** to the positive forms.
- N is realized as either n or m.

Replace Rules

A Motivating Example

Consider the following pairs:

Example

accurate inaccurate	adequate inadequate	balanced imbalanced	competent incompetent
definite indefinite	finite infinite	mature immature	nutrition innutrition
patience impatience	possible impossible	sane insane	tractable intractable

- The negative forms are constructed by adding the abstract morpheme **iN** to the positive forms.
- N is realized as either n or m.

- Replace rules are an extremely powerful extension of the regular expression metalanguage.
- The simplest replace rule is of the form
  `upper -> lower`
- Its denotation is the relation which maps string to themselves, with the exception that an occurrence of `upper` in the input string is replaced by `lower`.
- For example `N -> n`

```
xfst[0]: read regex N -> n;
xfst[1]: apply down iNcorrect
incorrect
xfst[2]: apply down iNperfect
inperfect
```

- Note that the rule itself compiles into an FST

- In order to get `imperfect` we need a rule like `N -> m` but this will yield wrong results (e.g. imcorrect).
- So we need to put context conditions on the rule.
- Conditional replace rules include left and/or right contexts.

```
upper -> lower || leftcontext _ rightcontext
```

- Its denotation is the relation which maps string to themselves, with the exception that an occurrence of `upper` *preceded by leftcontext and followed by rightcontext*, is replaced in the output by `lower`.

- A linguistically accurate way of handling these phenomena is to use two rules

  ```
  N -> m || _ [b|m|p]
  N -> n
  ```

  ensuring that their application is obligatory and that they are applied in the order given.

```
(read) regex <regexp> <semicolon>
regex <regexp> <semicolon>
(print) words
xfst[0]: regex [ d o g | c a t | h o r s e ] ;
xfst[1]: print words
horse
cat
dog
xfst[1]:
```

- The expression is read and compiled, and the network is pushed on the stack.
- The words of the top item are printed.
- The keywords `read` anmd `print` are optional

define <var> <regexp> <semicolon>

```
xfst[0]: define MyVar [ d o g | c a t | h o r s e ] ;
xfst[0]: regexp MyVar MyVar
xfst[1]: words
horsehorse
horsecat
horsedog
cathorse
catcat
catdog
doghorse
dogcat
dogdog
xfst[1]:
```

xfst Demonstration

apply up/down

```
(apply) up <word>
xfst[0]: regex [ d o g | c a t | h o r s e ] ;
xfst[1]: apply up dog
dog
xfst[1]: up pig
xfst[1]:
xfst[1]: down dog
dog
```

- The <word> is “looked up”
- The result of tranducing the word in an upward/downward direction is output.
- The keywords `read` and `print` are optional

xfst Demonstration

apply up/down from file

```
xfst[0]: regex < animals
Opening file animals...
420 bytes. 10 states, 11 arcs, 3 paths.
Closing file animals...
xfst[1]: up < wl
Opening file wl...

dog
dog

pig

horse
horse

cat
cat
```

xfst Demonstration

exponentiation operator

```
xfst[1]: regex a^2;
xfst[2]: words
aa
xfst[2]: regex a^{2,5};
228 bytes. 6 states, 5 arcs, 4 paths.
xfst[3]: words
aa
aaa
aaaa
aaaaa
xfst[4]:
```

```
(print) net
xfst[2]: regex a^{2,5};
228 bytes. 6 states, 5 arcs, 4 paths.
xfst[3]: net
Sigma: a
Size: 1
Net:
Flags: deterministic, pruned, minimized, epsilon_free, loop_free
Arity: 1
s0:   a -> s1.
s1:   a -> fs2.
fs2:  a -> fs3.
fs3:  a -> fs4.
fs4:  a -> fs5.
fs5:  (no arcs)
```

```
xfst[0]: regex a|b|c;
xfst[1]: regex b|c;
xfst[2]: regex a|b;
xfst[3]: intersect
xfst[1]: words
b
xfst[1]: regex [a|b|c] & [a|c] & [b|a];
xfst[2]: words
a
xfst[2]: union
xfst[1]: words
b
a
xfst[1]:
```

xfst Demonstration

cross product - possible abbreviations

The following are all equivalent

```
[[d o g] .x. [c h i e n]] |
[[c a t] .x. [c h a t]] |
[[h o r s e] .x. [c h e v a l]];

[{dog} .x. {chien}] |
[{cat} .x. {chat}] |
[{horse} .x. {cheval}];


{dog}   : {chien} |
{cat}   : {chat}  |
{horse} : {cheval};
```

```
xfst[0]: regex {dog}:{chien};
268 bytes. 6 states, 5 arcs, 1 path.
xfst[1]: up chien
dog
xfst[1]: down dog
chien

xfst[1]: words
<d:c><o:h><g:i><0:e><0:n>
xfst[1]:
```

```
xfst[0]: regex {dog}:{chien};
268 bytes. 6 states, 5 arcs, 1 path.
xfst[1]: up chien
dog
xfst[1]: down dog
chien

xfst[1]: words
<d:c><o:h><g:i><0:e><0:n>
xfst[1]:
```

Class Exercise 1

- The verb "sing" has the forms "sang" and "sung"
- Write a regular expression which allows you to *look up* any of these forms and get "sing".
- Draw the corresponding FST

Class Exercise 2a

Design and compile a network which has the following behaviour

```
xfst[] up black
black
xfst[] up blacker
black
xfst[] up blackest
black
```

Class Exercise 2b

- Modify the network to handle the forms of "green".
- Modify the network to perform morphological analysis i.e. it should give the part of speech as well as the degree of comparison, if applicable

```
xfst[] up green
green+JJ
xfst[] up greener
green+JJ+CMP
xfst[] up greenest
green+JJ+SUP
```

Class Exercise 2c

- What happens when you add the word "blue"?
- To fix the problem you need to use replace rules together with the composition operation.
- e.g. part of rule for eliminating "e" when it comes before %+COMP

```
R1 e -> 0 || _ %+COMP ;
```

- Irregular Plurals: don't just add an "s" (in EN)
  - Extra Irregular Plurals: irregular form is *in addition to* regular plural. example: fish (sing/pl), fishes (pl)
  - Overriding Irregular Plurals: irregular form *replaces* regular plural. example: index (sg), indices (pl)

Example

Noun	Regular	Irregular
fish	fishes	fish
lexicon	lexicons	lexica
person	persons	people

Example

Noun	Regular	Irregular
sheep	sheeps	sheep
corpus	corpuses	corpora
index	indexes	indices

Handling Irregular Plurals

- First note that the following *overgenerates* and *undergenerates* with wrt to the linguistic phenomena[1].

```
define NOUNS {cat} | {fish} |{sheep};
define NUMBER %+Noun%+SG:0 | %+Noun%+PL:s;
regex NOUNS NUMBER
words;
```

- apply up cat/cats: correct
- apply up fish/fishs: undergenerates
- apply up sheep/sheeps: overgenerates

[1] NB: for the moment we ignore the misspelling of "fishes"

Handling Undergeneration

Extra Irregular Plurals

- In this case we need to add the fact that "fish" is both singular and plural.
- One way to do this is to simply add an extra path into the network and then add this using the union operation

```
define NOUNS {cat} | {fish} |{sheep};
define NUMBER %+Noun%+SG:0 | %+Noun%+PL:s;
define EXTRA {fish} %+Noun%+PL:0
regex [NOUNS NUMBER] | EXTRA
words;
```

- Now we also get that "fish" can be plural

Handling Overgeneration

To handle overgeneration, we can

1. Define a grammar which overgenerates regular plurals, i.e. adds "s" even when it shouldn't.
2. Use composition to filter out the overgenerated plurals
3. Use union to add the overriding irregular plurals

Handling Irregular Plurals

1. Define a grammar which overgenerates regular plurals, i.e. adds "s" even when it shouldn't.

```
define NOUNS {cat} | {fish} |{sheep};
define NUMBER %+Noun%+SG:0 | %+Noun%+PL:s;
define LEX [NOUNS NUMBER]
```

Handling Irregular Plurals

1. Define a grammar which overgenerates regular plurals, i.e. adds "s" even when it shouldn't.
2. Use composition to filter out the overgenerated plurals

```
define NOUNS {cat} | {fish} |{sheep};
define NUMBER %+Noun%+SG:0 | %+Noun%+PL:s;

define OVERRIDING {sheep} %+Noun%+PL:0;
define FILTER OVERRIDING.u;

define LEX [NOUNS NUMBER] | OVERRIDING
define FILTEREDLEX ~FILTER .O. LEX
```

Handling Irregular Plurals

1. Define a grammar which overgenerates regular plurals, i.e. adds "s" even when it shouldn't.
2. Use composition to filter out the overgenerated plurals
3. Use union to add the overriding irregular plurals

```
define NOUNS {cat} | {fish} |{sheep};
define NUMBER %+Noun%+SG:0 | %+Noun%+PL:s;

define OVERRIDING {sheep} %+Noun%+PL:0;
define FILTER OVERRIDING.u;

define LEX [NOUNS NUMBER] | OVERRIDING
define FILTEREDLEX ~FILTER .O. LEX

define EXTRA {fish} %+Noun%+PL:0;

define GOODLEX FILTEREDLEX | EXTRA
```

xfst Programming Idioms

- We have seen how irregular plurals can override unwanted regular plurals.
- To achieve this we used an “idiom" that combines two things:
  - upperside filtering
  - union
- It turns out to be such a useful idiom that it has been packaged into a single operator which is part of the `xfst` language.

Priority Union

- The priority union operator is written `L .P. R`
- `L .P. R` is not symmetrical
- The result of `L .P. R` is a union of L and R except that whenever `L` and `R` have the same string on the upper side, the path in `L` takes priority.

Priority Union

```
define L a:1 | b:2 | c:3;
define R a:3 | c:4 | d:5;
regex L .P. R;
words
```

Use of Priority Union

```
define NOUNS {cat} | {fish} |{sheep};
define NUMBER %+Noun%+SG:0 | %+Noun%+PL:s;
define EXTRA {fish} %+Noun%+PL:0;
define OVERRIDING {sheep} %+Noun%+PL:0;

define LEX [NOUNS NUMBER] | EXTRA | OVERRIDING;
define FILTEREDLEX OVERRIDING .P. LEX;
regex FILTEREDLEX | OVERRIDING;
```

There are three main phenomena of interest:

Non-Concatenative Morphology

- Fixed Length Reduplication
- Full Stem Reduplication
- Stem Interdigitation

Non-Concatenative Morphology

- Fixed Length Reduplication
- Full Stem Reduplication
- Stem Interdigitation

Non-Concatenative Morphology

- Fixed Length Reduplication
- Full Stem Reduplication
- Stem Interdigitation

Non-Concatenative Morphology

- Fixed Length Reduplication
- Full Stem Reduplication
- Stem Interdigitation

Fixed Length Reduplication
Tagalog

ROOT	CV+ROOT	GLOSS
pili	pipili	choose
tahi	tatahi	sew
kuha	kukuha	take

Full Stem Reduplication
Malay

ROOT	GLOSS	REDUPLICATION	GLOSS
anak	child	anak-anak	children
lembu	cow	lembu-lembu	cows
buku	book	buku-buku	books
basikal	bicycle	basikal-basikal	bicycles

Stem Interdigitation

Maltese, Arabic, Hebrew

- Stems are composed of
  - root consisting of consonants such as ktb
  - pattern consisting of vowels such as _i_e_ and slots into which consonants are inserted
- Root and pattern are “interdigitated” to form stems like “kiteb" and “ktieb"
- NB. The role of vowels in Arabic is much more complex and also systematic than in Maltese.

Interdigitation using Compile Replace

```
list C b t y k l m n f w r z d s
list V a i u e
regex {ktb} .m>. {CVCVC} .<m. [i|e]+;
words
```