Formal Grammar III: Extended BNF, Regular Expressions

CAS CS 320: Principles of Programming Languages

Thursday, March 7, 2024

Administrivia

- Homework 5 due Friday, Mar 8 (tomorrow), by 11:59 pm.
- Homework 6 posted Friday, Mar 8 (tomorrow), and due after Spring Break.
- Grading of midterm should be completed by the end of the day today.

REMINDERS FROM PRECEDING TWO LECTURES

REMINDERS FROM PRECEDING TWO LECTURES

Concepts we have already introduced:

BNF grammars, context-free grammars

rules, production rules

derivations

terminals, non-terminals

sentential forms, sentences

ambiguity, how to avoid it

associativity, fixity of ops, precedence

BNF example

A grammar is defined by a set of terminals (tokens), a set of nonterminals, a designated **nonterminal start symbol**, and a finite nonempty set of rules

```
<sentence>     ::= <noun-phrase><verb-phrase>.
<noun-phrase>  ::= <article><noun>
<article>      ::= a | an | the
<noun>         ::= man | apple | worm | penguin
<verb-phrase>  ::= <verb> | <verb><noun-phrase>
<verb>         ::= eats | throws | sees | is
```

Derivations using BNF

A derivation is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)

```
<sentence>    ::= <noun-phrase><verb-phrase>.
<noun-phrase> ::= <article><noun>
<article>     ::= a | an | the
<noun>        ::= man | apple | worm | penguin
<verb-phrase> ::= <verb> | <verb><noun-phrase>
<verb>        ::= eats | throws | sees | is
```

A derivation example

```
<sentence> => <noun-phrase><verb-phrase>.
           => <article><noun><verb-phrase>.
           => the<noun><verb-phrase>.
           => the man <verb-phrase>.
           => the man <verb><noun-phrase>.
           => the man eats <noun-phrase>.
           => the man eats <article><noun>.
           => the man eats the <noun>.
           => the man eats the apple.
```

Derivations and sentences

- Every string of symbols in the derivation is a sentential form.
- A sentence is a sentential form that has only terminal symbols.
- A leftmost derivation is one in which the leftmost nonterminal in each sentential form is the one that is expanded.
- A derivation may be either leftmost or rightmost (or something else)

Another BNF example

```
<program> ::= <stmts>
<stmts>   ::= <stmt> | <stmt> ; <stmts>
<stmt>    ::= <var> = <expr>
<var>     ::= a | b | c | d
<expr>    ::= <term> + <term> | <term> - <term>
<term>    ::= <var> | const
```

Note: grammar rules can be recursive.

A derivation example

```
<program> => <stmts>
          => <stmt>
          => <var> = <expr>
          => a = <expr>
          => a = <term> + <term>
          => a = <var> + <term>
          => a = b + <term>
          => a = b + const
```

Generator vs Recognizer

```
<program> ::= <stmts>
<stmts> ::= <stmt> | <stmt> ; <stmts>
<stmt> ::= <var> = <expr>
<var> ::= a | b | c | d
<expr> ::= <term> + <term> | <term> - <term>
<term> ::= <var> | const
```

Recognize a sentence

```
a = b + const   <=
<var> = b + const   <=
<var> = <var> + const   <=
<var> = <term> + const   <=
<var> = <term> + <term>   <=
<var> = <expr>   <=
<stmt>   <=
<stmts>   <=   <program>
```

Generate a sentence

```
<program>   =>   <stmts>
            =>   <stmt>
            =>   <var> = <expr>
            =>   a = <expr>
            =>   a = <term> + <term>
            =>   a = <var> + <term>
            =>   a = b + <term>
            =>   a = b + const
```

NEW MATERIAL FOR THIS LECTURE

1. **Extended BNF**

2. **Regular Grammars, Regular Expressions**

Extended BNF, abbreviated as EBNF

Syntactic sugar: **EBNF does not extend BNF's expressive power, but does make it easier to use.**

Extended BNF, abbreviated as EBNF

Syntactic sugar: **EBNF does not extend BNF's expressive power, but does make it easier to use.**

Optional parts **are placed in square brackets [ ]**

Alternative parts **are placed in parentheses ( ) and separated with vertical bars |**

Repeated parts **(0 or more) are placed in braces { }**

Extended BNF, abbreviated as EBNF

Example of an *optional part*

BNF

<if_stm> ::= if <expr> <stm> | if <expr> <stm> else <stm>

EBNF – we can use square brackets:

<if_stm> ::= if <expr> <stm> [else <stm>]

Extended BNF, abbreviated as EBNF

Example of an *alternative part*

BNF

<term> ::= <term> + <const> | <term> - <const>

EBNF – we can use parentheses with vertical bars:

<term> ::= <term> (+ | -) <const>

Extended BNF, abbreviated as EBNF

Example of a *repeated part*

BNF

<ident> ::= <letter> | <ident> <letter>

EBNF – we can use braces:

<ident> ::= <letter> { <letter> }

Extended BNF, abbreviated as EBNF

Example of a *repeated part*

BNF

<ident> ::= <letter> | <ident> <letter>

EBNF – we can use braces:

<ident> ::= <letter> { <letter> }

BNF

<ident> ::= <letter> | <ident> <letter> | <ident> <digit>

EBNF – we can use braces with vertical bars:

<ident> ::= <letter> { <letter> | <digit> }

Extended BNF, abbreviated as EBNF

Example with *alternative* and *repeated parts*

Suppose we have the following BNF:

<expr> ::= <expr> + <term> | <expr> - <term> | <term>

<term> ::= <term> * <factor> | <term> / <factor> | <factor>

We can reformulate it as the following EBNF:

<expr> ::= <term> {(+ | -) <term>}

<term> ::= <factor> {(* | /) <factor>}

Regular Grammars

Only three kinds of rules:

<non-terminal> ::= terminal

<non-terminal> ::= terminal <non-terminal>

<non-terminal> ::= empty

This is a right linear grammar.

Regular expressions

A compact way to describe regular grammars:

- A terminal is a regular expression
- The or | of two expressions is a regular expression describing two alternatives.
- The grouping (_) of expressions is a regular expression describing sequencing of symbols
- The quantification * of a regular expression is a regular expression describing zero or more occurrence of the same regular expression

Regular expressions

A compact way to describe regular grammars:

- A terminal is a regular expression
- The or | of two expressions is a regular expression describing two alternatives.
- The grouping (_) of expressions is a regular expression describing sequencing of symbols
- The quantification * of a regular expression is a regular expression describing zero or more occurrence of the same regular expression

Also known as Kleene star

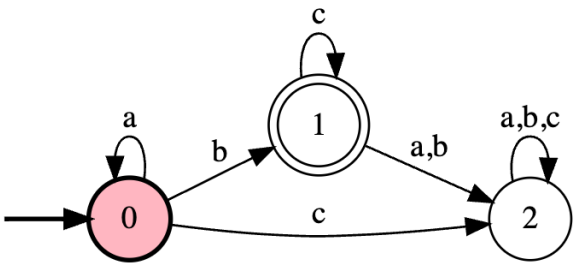
Regular grammars - example

```
<S>  ::=  a<S>
<S>  ::=  b<A>
<A>  ::=  ε
<A>  ::=  c<A>
```

We can describe the grammar above by the following expression.

a*bc*

Regular expressions



Finite state automata

// =

Regular grammars = Regular expressions

```
<S> ::= a<S>
<S> ::= b<A>
<A> ::= Ɛ
<A> ::= c<A>
```

a*bc*

All equivalent in terms of language recognized.
Not equivalent in ease of use.

Regular expressions vs context free grammars

- Regular expressions cannot express everything we can express with context free grammar. (example: matching parentheses)
- A regular expression recognizer/generator is much simpler to implement than a parser.
- Regular expressions give potentially infinite vocabularies.

Regular expression

a*bc*

Star means repetition

Examples recognized:

b	0 repetitions of a and c
ab	1 repetition of a, 0 repetition of c
abc	1 repetition of a, 1 repetition of c
aab	2 repetitions of a, 0 repetition of c
aaaaaaabccccccc	7 repetitions of a, 6 repetitions of c

Regular expression

aaa*bbb*

Star means repetition

Examples recognized:

aabb

aaaaabb

aabbbb

aaabbb

Regular expression

a((bc)|(cb))*d

Star means repetition

Examples recognized:

abcd

abccbd

acbbcd

abccbbcd

Regular expression

(a|b)*

Examples recognized:

aaaa

abbba

abababa

bababbba

a*|b*

Examples recognized:

a

bbb

aaaaaa

bbbbbbb

Regular expression

(a|b)*

Examples recognized:

aaaa

abbba

abababa

bababbba

a*|b*

Examples recognized:

a

bbb

aaaaaa

bbbbbbb

Not equivalent. a*|b* does not accept sentences containing both a and b.

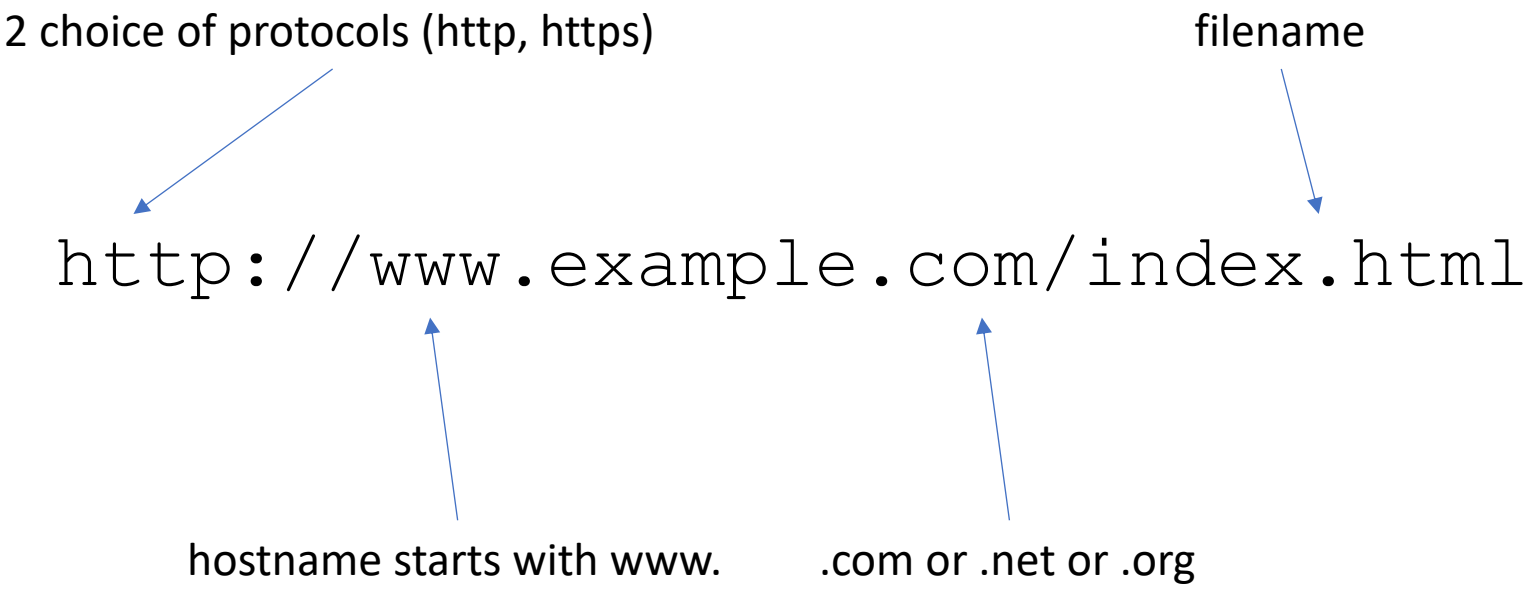
Designing Regular expression

We are to check if a string is a valid URL.

How can we design a regular expression to recognize valid URLs?

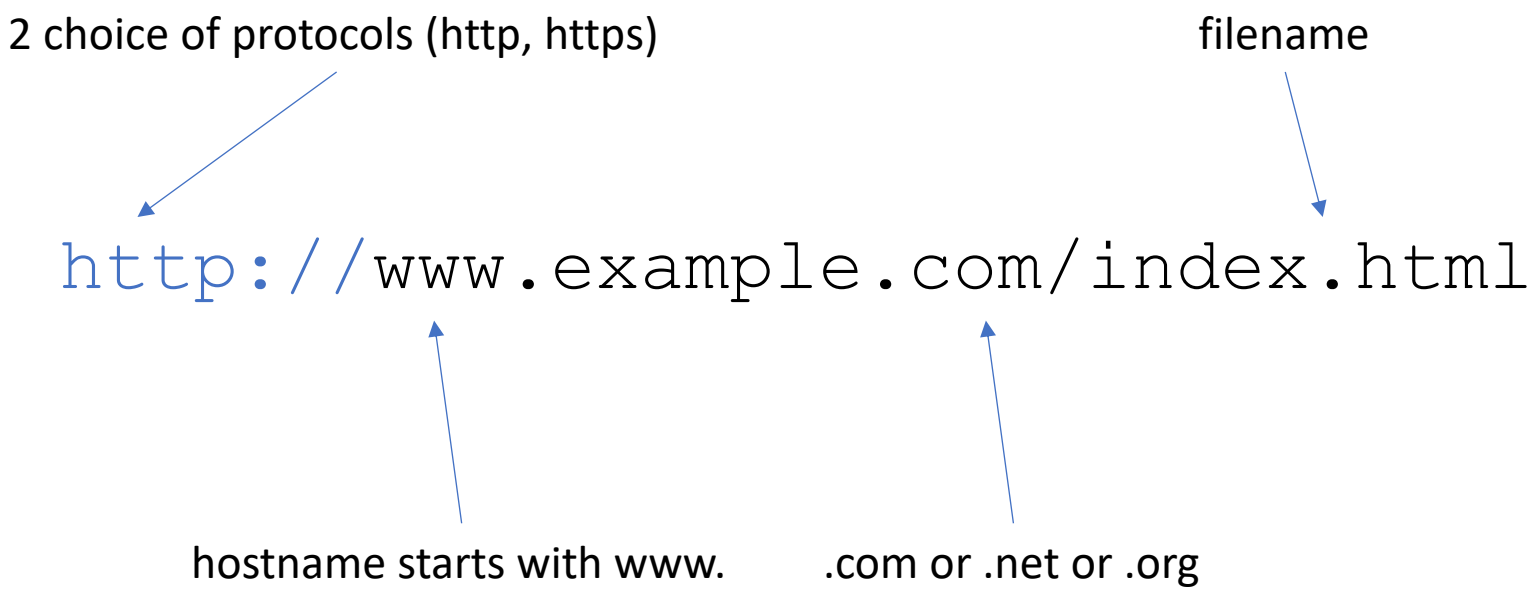
`http://www.example.com/index.html`

Designing Regular expression



Designing Regular expression

http(ε|s)://



Designing Regular expression

http(ε|s)://www.[a-z]*

Notation for any character from a to z

2 choice of protocols (http, https)

filename

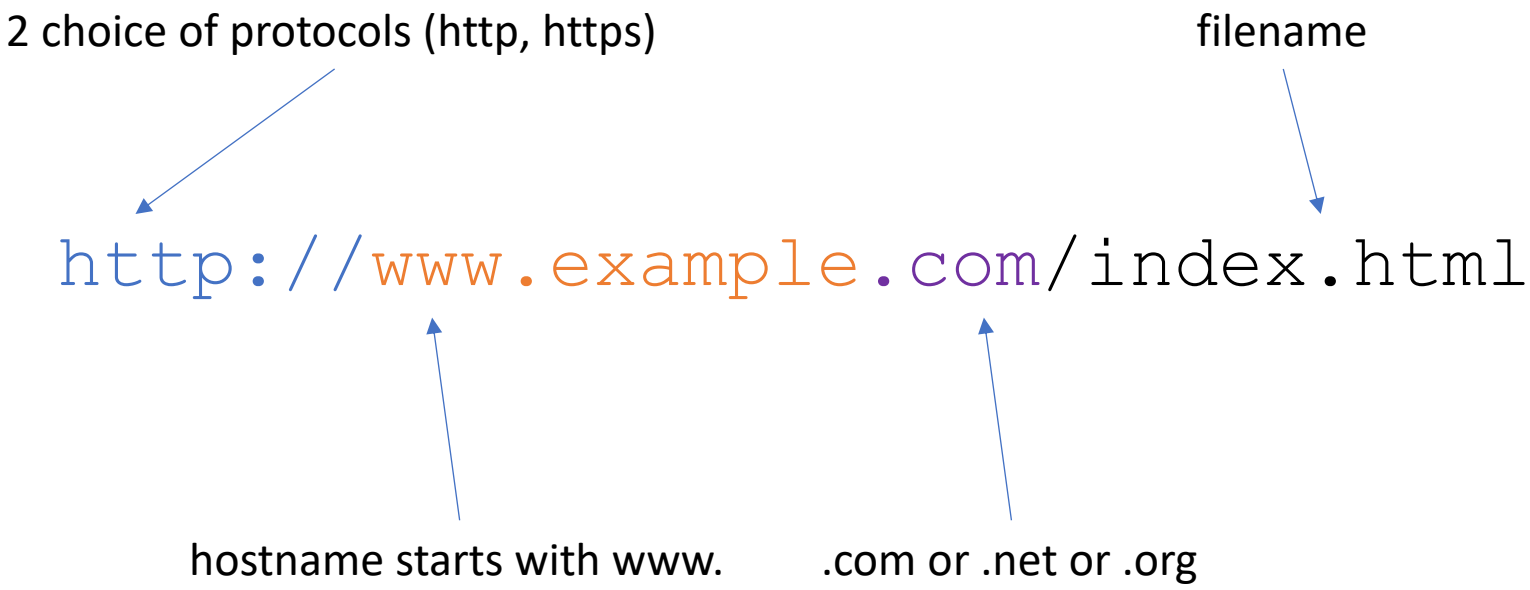
http://www.example.com/index.html

hostname starts with www.

.com or .net or .org

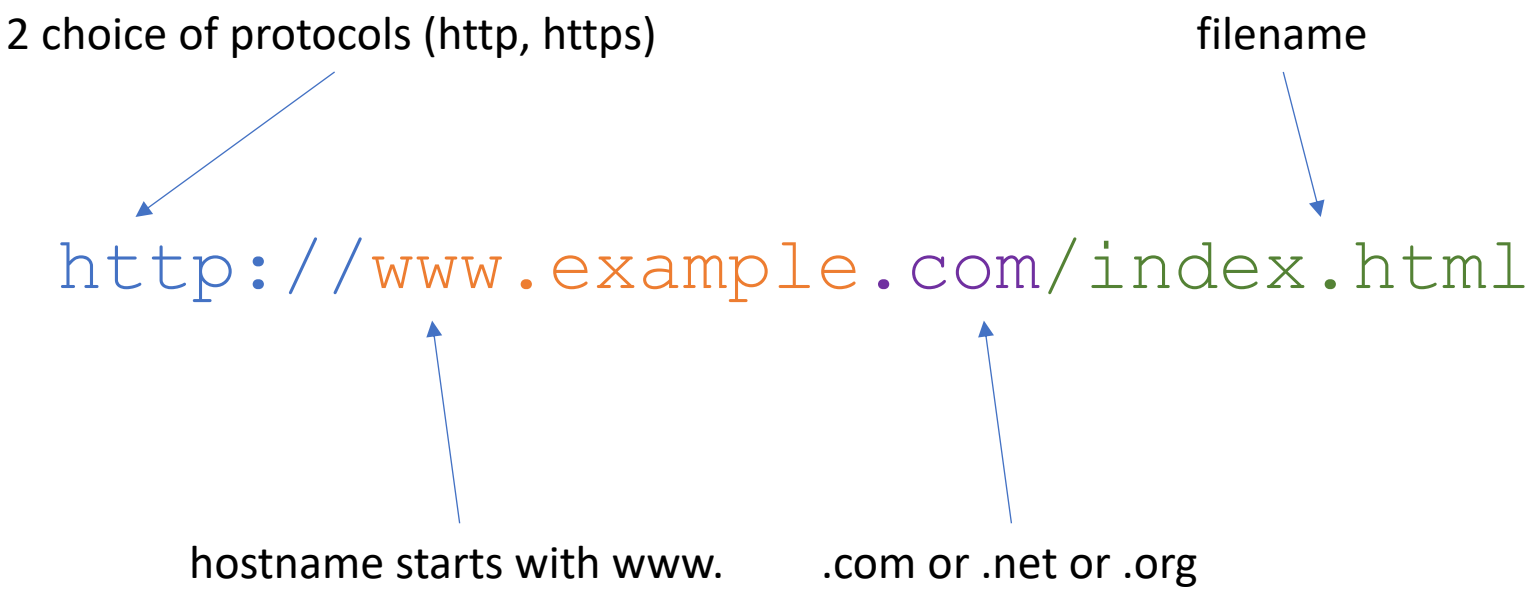
Designing Regular expression

http(ε|s)://www.[a-z]*.((com)|(net)|(org))



Designing Regular expression

http(ε|s)://www.[a-z]*.((com)|(net)|(org))(/|[a-z]|.)*



Designing Regular expression

http(ε|s)://www.[a-z]*.((com)|(net)|(org))(/|[a-z]|.)*

Which other URLs are recognized by this regular expression?

https://www.google.com
https://mail.google.com/
https://www.youtube.com/feed/subscriptions
https://www.youtube.com/c/F1
https://www.bu.edu
https://www.amazon.com/

Designing Regular expression

http(ε|s)://www.[a-z]*.((com)|(net)|(org))(/|[a-z]|.)*

Which other URLs are recognized by this regular expression?

- ☑ https://www.google.com
- ☒ https://mail.google.com/
- ☑ https://www.youtube.com/feed/subscriptions
- ☒ https://www.youtube.com/c/F1
- ☒ https://www.bu.edu
- ☑ https://www.amazon.com/

Our regular expression is too restrictive. Only recognizes simple URLs.
Regular expression for real-world URL is much more complex.

(THIS PAGE INTENTIONALLY LEFT BLANK)