COP5556 Programming Language Principles

Parsing 1

Reading

- The rest of Scott Chapter 2
  - Including section 2.3.5 on the online supplement
  - Skim 2.3.3 Bottom-Up Parsing

- Recall: first step of language translation is lexical analysis
  - Converts sequence of characters into a sequence of tokens
  - Lexical structure usually specified with regular expressions
- Second step of language translation is parsing
  - Recognizes legal phrases (order of tokens)
  - Constructs AST

New formalism

- Regular expressions are not recursive, thus cannot specify nested constructs

- Example: language of expressions

  (x + y) + (w + z)

  - is an expression with
  - two subexpressions, (x + y) and (w + z)

  - each with two subexpressions x and y, and w and z

- The language of expressions cannot be specified with regular expressions.

- Instead, we specify the language with a grammar using BNF (Backus-Naur Form) or EBNF (extended BNF) notation.

- BNF allows us to specify context-free grammars

Context Free Grammar

- A grammar is a tuple (Σ, N, P, S) where
  - Σ is a set specifying the alphabet of tokens (also called terminal symbols).
  - N is the set of non-terminal symbols.
    - non-terminals are variables that denote sets of sentences.
    - impose a hierarchical structure on sentences.
  - P is a set of productions (substitution rules).
  - S is the start symbol where S is in N.

Benefits of (E)BNF notation

- Compact and precise specification that allows an infinite number of sentences to be specified with a finite definition
- Allows systematic or automatic generation of efficient parsers for certain types of grammars
- Provides a framework for specifying semantics. We will see this later.
- Allows formal reasoning about languages.

Example

$S ::= \varepsilon$

$S ::= (S)S$

- Σ, N are left implicit
  - but Σ = {(,)}, N = {S}
- Recursive: S defined in terms of itself
- As expected, ε is the empty sequence

- What language does this grammar generate?

Example

$S ::= \varepsilon$
$S ::= (S)S$

- Σ, N are left implicit
  - but Σ = {(,)}, N = {S}

- Note recursion—S defined in terms of itself

- As expected, ε is the empty sequence

- The set of strings of balance parentheses

EBNF Notation

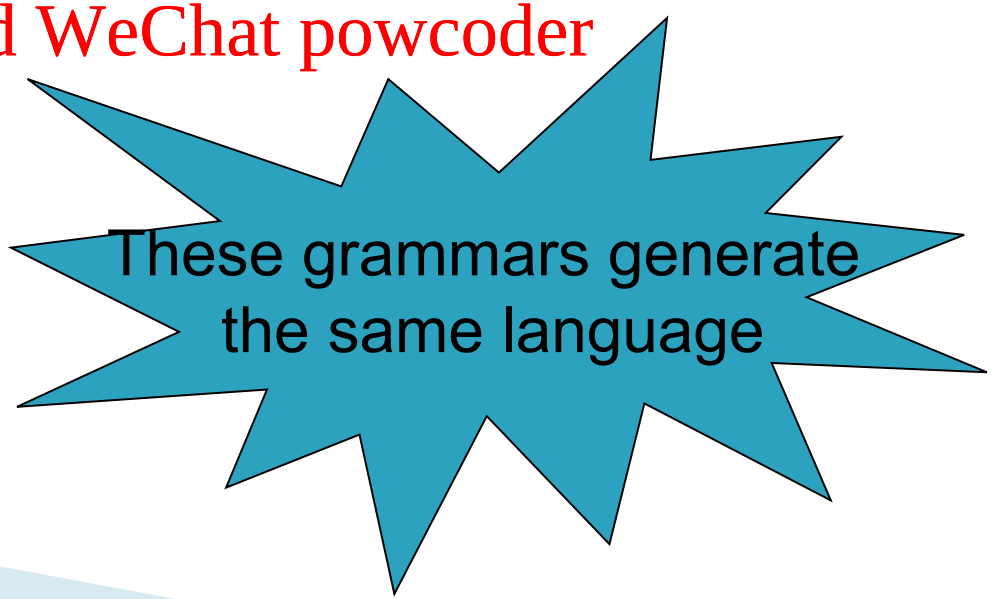
- Pure BNF only allows productions only of the form $\sigma 1 ::= \sigma 2$, where $\sigma 1$ and $\sigma 2$ are strings of terminal and non-terminal symbols
- EBNF (extended BNF) adds |, *, + from RE
- Example
  - BNF

    $A ::= \alpha\ B\ \gamma$

    $B ::= \beta\ B$

    $B ::= \varepsilon$
  - EBNF

    $A ::= \alpha\ \beta^*\ \gamma$



EBNF Notation

- Pure BNF only allows productions only of the form $\sigma 1 ::= \sigma 2$, where $\sigma 1$ and $\sigma 2$ are strings of terminal and non-terminal symbols
- EBNF (extended BNF) adds |, *, + from RE
- Example
  - BNF

    $A ::= \alpha\ B\ \gamma$

    $B ::= \beta\ B$

    $B ::= \varepsilon$
  - EBNF

    $A ::= \alpha\ \beta^*\ \gamma$

Recursion in parser

Iteration in parser

Example

expr ::= *factor* | *expr* **op** *factor*
factor ::= **ident** | **numlit** | **(** *expr* **)**

ident, numlit, op are tokens which were defined earlier.

```
<token> ::= <identifier> | <numlit> | <op> | ( | )

<numlit> ::= <nonzero digit> <digit>* | 0
<digit>  ::=  0 | <nonzero digit>
<nonzero digit> ::=  1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
9

<identifier> ::=
        <identifier start> <identifier part>*
<identifier start> ::=  A .. Z | a..z | $ | _
<identifier part> ::=
        <identifier start> | <digit>

<op> ::=   + | == | *
```

expr ::= *factor* | *expr* **op** *factor*
factor ::= **ident** | **numlit** | **(** *expr* **)**

Some expressions:

x
34
s + 2
s==3+4
(4+5)==9
(1 * (2+3) == 4) + 5
abc == 34 * 4

expr ::= *factor* | *expr* **op** *factor*
f*actor* ::= **ident** | **numlit** | **(** *expr* **)**

What is Σ?

expr ::= *factor* | *expr* **op** *factor*
f*actor* ::= **ident** | **numlit** | **(** *expr* **)**

Σ = {**ident**, **numlit**, **op**, **(** , **)** }

- Derivation of expressions

expr ::= *factor* | *expr* **op** *factor*
f*actor* ::= **ident** | **numlit** | **(** *expr* **)**

x

expr →
factor →
ident(x)

expr ::= *factor* | *expr* **op** *factor*
f*actor* ::= **ident** | **numlit** | **(** *expr* **)**

34

expr →
factor →
numlit(34)

expr ::= *factor* | *expr* **op** *factor*

f*actor* ::= **ident** | **numlit** | **(** *expr* **)**

s+2

expr →

expr op(+) factor →

factor op(+) factor →

ident(s) op(+) factor →

ident(s) op(+) numlit(2)

Left as exercise

s==3+4

(4+5)==9

(1 * (2+3) == 4) + 5

abc == 34 * 4

Non-context free grammars

- The rule αAβ ::= ασβ
  - only allows A to be replaced by σ when it appears between α and β.
  - contextual constraint on the substitution of A makes this grammar NOT context-free.
- vs a context-free grammar, where
  - all the productions have the form A ::= ..... with a single non-terminal on the left side.
  - This allows A to be replaced anywhere it appears.

- Programming languages are NOT context free.
  - But we use context-free grammars anyway.

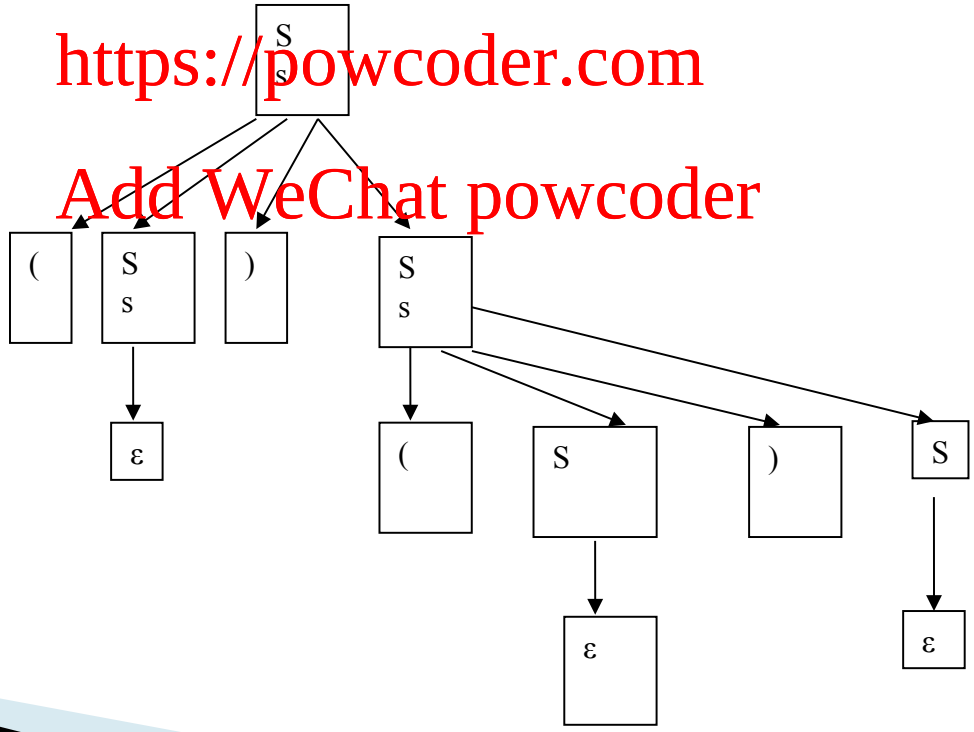
  - A context-free description of the grammar is a useful starting point for implementation of translators.
  - The context constraints are specified and checked using different techniques.

Parsing

- Grammars give rules for generating sentences in a language
- Parsing is the process of recognizing the language (and determining its structure)
- Parse trees are a way of representing the structure of a sentence
  - start symbol at the root
  - non-terminals at interior nodes
  - terminals as leaves

Example: parse tree

- Sentence ()() from grammar S ::= (S)S | ε.



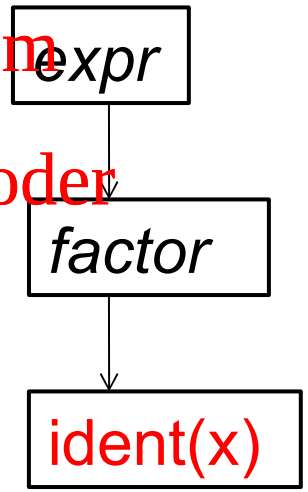
Example: parse tree

expr ::= *factor* | *expr* **op** *factor*

f*actor* ::= **ident** | **numlit** | **(** *expr* **)**

x

expr →
factor →
ident(x)

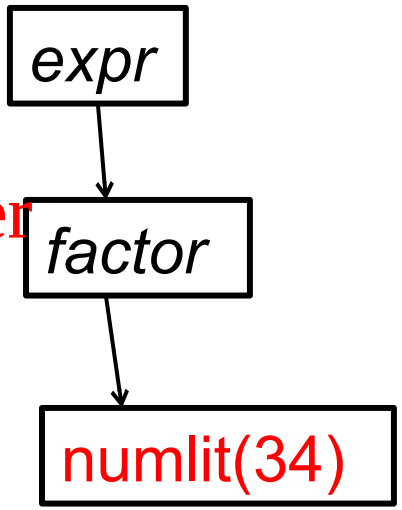


Example: parse tree

expr ::= *factor* | *expr* **op** *factor*
factor ::= **ident** | **numlit** | **(** *expr* **)**

34

expr →
factor →
numlit(34)

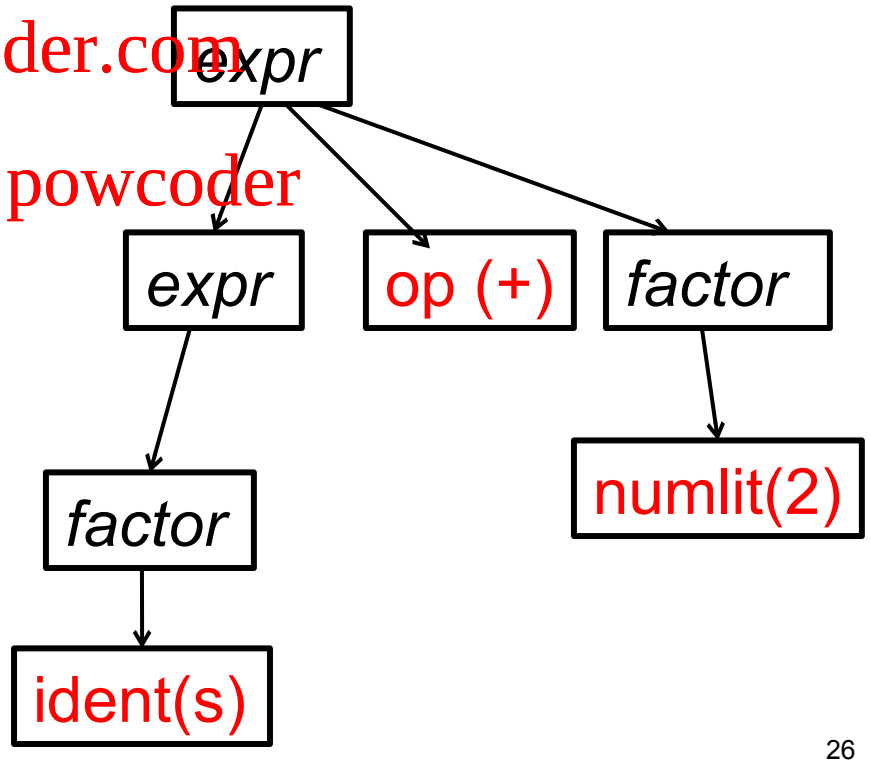


Example: Parse tree

expr ::= *factor* | *expr* **op** *factor*
factor ::= **ident** | **numlit** | **(** *expr* **)**

s+2

expr →
expr op(+) factor →
factor op(+) factor →
ident(s) op(+) factor →
ident(s) op(+) numlit(2)



Ambiguous Grammar

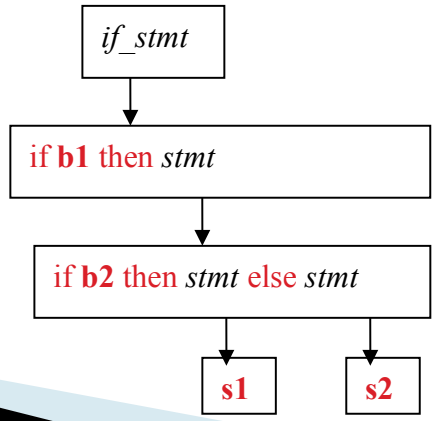
- Admits more than one parse tree for a sentence.
- Example

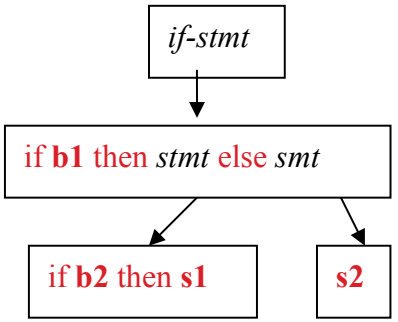
  stmt ::= *if_stmt* | (other non-if statements)...

  if_stmt ::= if *boolean_expr* then *stmt*

  if_stmt ::= if *boolean_expr* then *stmt* else *stmt*

if b1 then if b2 then s1 else s2





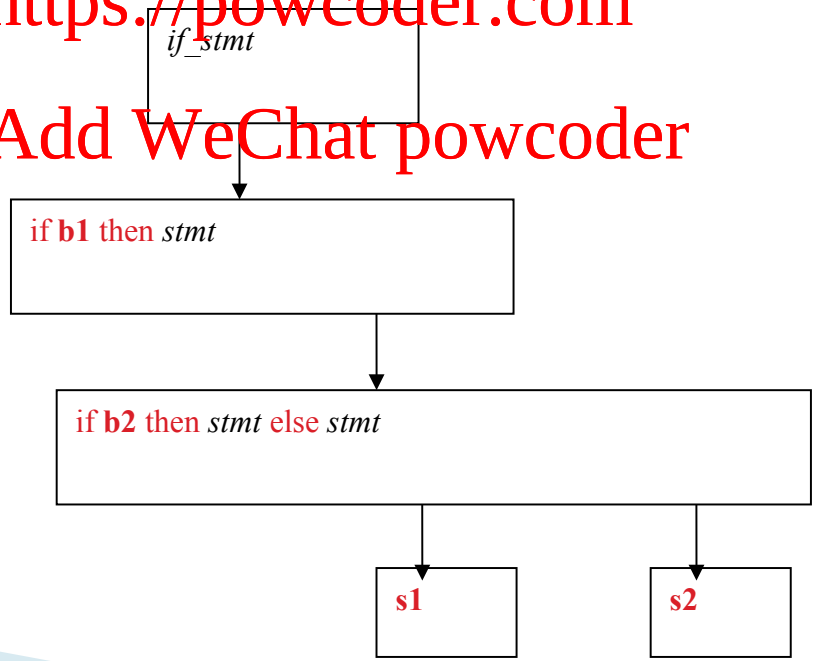
Dealing with ambiguous grammars

- Without changing the programming language being specified:
  - Use disambiguating rule to add "hack" to parser.
    - else is matched with nearest previously unmatched if.
  - or, modify the grammar so it is non-ambiguous
    - Resulting grammar must generate the same language
    - Grammars are not unique: many grammars may specify the same languages
- Change the language (if you can)
  - Often this improves the language

Example: Ambiguous grammar with disambiguating rule

else is matched with nearest previously unmatched if

if b1 then if b2 then s1 else s2



Example: modify the grammar

- Distinguish between matched and unmatched statements
- You can't have an unmatched statement immediately before an else

stmt ::= *matched* | *unmatched*

matched ::=
  if *boolean_expr* then *matched* else *matched*
  | ....(other non-if stmt)

unmatched ::=
  if *boolean_expr* then *stmt*
  | if *boolean_expr* then *matched* else *unmatched*

stmt ::= *matched* | *unmatched*

matched ::= if *boolean_expr* then *matched* else *matched*

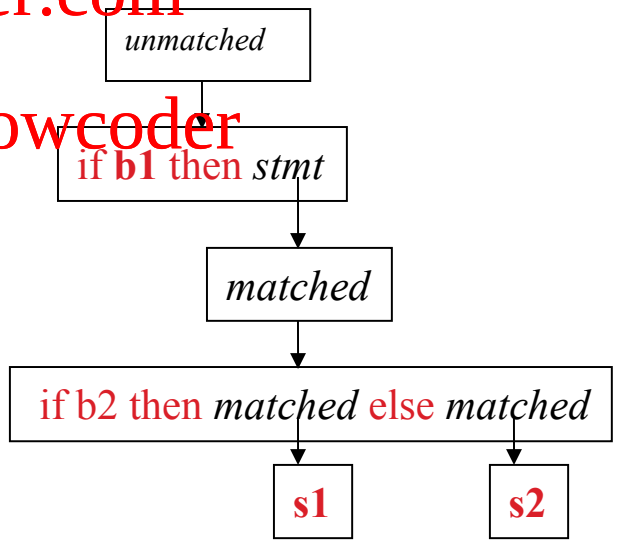
| ....(other non-if stmt)

unmatched ::= if *boolean_expr* then *stmt*

| if *boolean_expr* then *matched* else *unmatched*

if b1 then if b2 then s1 else s2

(only one works)



Example: modify the language

- Change the language to explicitly terminate if statements.

stmt ::= if *boolean_expr* then *stmt* endif
| if *boolean_expr* then *stmt* else *stmt* endif
| other non-if statements

Allowed sentences

if b1 then if b2 then s1 else s2 endif endif

if b1 then if b2 then s1 endif else s2 endif

Parsing Complexity

- The difficulty of recognition depends on the complexity of the grammar.
- Grammars can be classified according to the class of parsing algorithms that can recognize the language.
- There are $O(n^3)$ algorithms that work for any context free grammar
- For useful subsets of CFG, there are parsers that are linear.
  - LL (left-to-right, leftmost derivation) top-down (often hand written)
  - LR (left-to-right, rightmost derivation) bottom-up (usually generated)

- So far,
  - Lexical analysis
    - can specify tokens with RE
    - recognize tokens with DFA implemented by a scanner
    - scanners also keep track of position of tokens in the source text and handle white space, keywords, and comments.

- Phrase structure
  - use EBNF notation to specify context-free grammar
    - in contrast to REs, can be recursive
    - allows nested constructs to be specified
  - parsers recognize phrases
  - multiple CFGs can specify the same language; we try to choose one that is most appropriate for our purposes
    - clear to human reader
    - can be parsed by a desirable parsing algorithm
  - parse trees illustrate the structure of a phrase
    - an ambiguous grammar admits multiple parse trees
    - Options for dealing with ambiguity
      - Impose a disambiguating rule
      - Eliminate ambiguity by using a different grammar that generates the same language but is not ambiguous.
      - If possible, change the language.

Approaches to parsing

- bottom-up
  - we will only briefly cover this
- top-down (recursive descent)
  - we will focus on this and use in our project
  - characteristics of grammars that allow top-down parsing
    - LL grammars
    - FIRST, FOLLOW, and PREDICT sets
  - transformations on grammars

Bottom-up parsing

- Scan, looking for leftmost substring that matches r.h.s (right hand side) of some production, replace with l.h.s. (left hand side)
- Repeat.
- If the start symbol is obtained, the sentence is in grammar.

- Usually, bottom up parsers are generated by a tool.

Example: Bottom up parsing

S ::= AB

A ::= x | y

B := z | w

String to parse: xw

Example: Bottom up parsing (2)

S ::= AB

A ::= x | y

B := z | w

String to parse: xw

xw

←

Aw

Example: Bottom up parsing (3)

S ::= AB
A ::= x | y
B := z | w

String to parse: xw

xw
←
Aw
←
AB

Example: Bottom up parsing (4)

S ::= AB
A ::= x | y
B := z | w

String to parse: xw

xw
←

Aw
←
AB
←
S This is the start symbol, so the string is in the language

Example 2: Bottom-up parsing (1)

S ::= aABe
A ::= Abc | b
B ::= d

String to recognize: abbcde

Example 2: Bottom-up parsing (2)

S ::= aABe
A ::= Abc | b
B ::= d

String to recognize: abbcde

- Scan, looking for substring that matches the right side of some production.
- b and d qualify.
- Choose leftmost one, b, and replace it by A.

  abbcde

  ←

  aAbcde

Example 2: Bottom-up parsing (3)

S ::= aABe
A ::= Abc | b
B ::= d

String to recognize: abbcde

abbcde
←
aAbcde

- Now substrings Abc, b, and d match a r.h.s. Abc is leftmost, so replace Abc by A

aAbcde
←
aAde

Example 2: Bottom-up parsing (4)

S ::= aABe
A ::= Abc | b
B ::= d

String to recognize: abbcde

abbcde
←
aAbcde
←
aAde

- replace d with B

aAde
←
aABe

Example 2: Bottom-up parsing (5)

S ::= aABe
A ::= Abc | b
B ::= d

String to recognize: abbcde

abbcde
←
aAbcde
←
aAde
←
aABe
←
S

Top down parsing

- Start with start symbol in language
- At each step, replace a nonterminal symbol with one of its productions, trying to match prefixes in the sentence
- Remove matching prefixes
- If the resulting string is empty, then the sentence is in the grammar

Example: Top-down parsing (1)

S ::= AB

A ::= x | y

B := z | w

String to parse: xw

Example: Top-down parsing (2)

S ::= AB

A ::= x | y

B := z | w

String to parse xw

S xw

Example: Top-down parsing (3)

S ::= AB

A ::= x | y

B := z | w

String to parse xw

S xw

→

AB xw (no match, just replace S with its r.h.s)

Example: Top-down parsing (4)

S ::= AB
A ::= x | y
B := z | w

String to parse xw

S xw

→

AB xw (replace A with one of its alternatives)

→

xB xw

Example: Top-down parsing (5)

S ::= AB
A ::= x | y
B := z | w

String to parse xw

S xw
→
AB xw
→
xB xw (x is a common prefix)
→
B w (remove x from both sides)

Example: Top-down parsing (6)

S ::= AB
A ::= x | y
B := z | w

String to parse xw

S xw
→
AB xw
→
xB xw
→
B w (replace B with one of its alternatives)
→
w w

Example: Top-down parsing (7)

S ::= AB
A ::= x | y
B := z | w

String to parse xw

S	xw
→	
AB	xw
→	
xB	xw
→	
B	w
→	
w	w (delete common prefix)
→	
ε	ε

OK String is legal

Grammars and Parsing

- Recall grammars generate, parsers recognize
- $O(n^3)$ parsing algorithms for any context-free grammar exist, but we want a linear algorithm

- Goal: identify special classes of context-free grammars that can be parsed with linear algorithms
  - LL grammars (top down parsing alg.)
  - LR grammars (bottom up parsing alg.)

- Now we will explore LL(1) grammars in more detail
  - Look at some motivating examples to see problems
  - State rules for grammars that rule out the problematic situations

Top-down parsing

S ::= A|B

A ::= xA | y

B ::= xB | z

String to parse: xxz

Top-down parsing

S xxz

S ::= A|B

A ::= xA | y

B ::= xB | z

String to parse: xxz

Top-down parsing

S ::= A|B
A ::= xA | y
B ::= xB | z

String to parse: xxz

S	xxz
→	
A	xxz

Top-down parsing

S ::= A|B

A ::= xA | y

B ::= xB | z

String to parse: xxz

S		xxz
→		
xA		xxz
→		
xA		xxz

Top-down parsing

S ::= A|B

A ::= xA | y

B ::= xB | z

String to parse: xxz

S		xxz
	→	
A		xxz
	→	
xA		xxz
→ consume x		
A		xz

Top-down parsing

S ::= A|B
A ::= xA | y
B ::= xB | z

String to parse: xxz

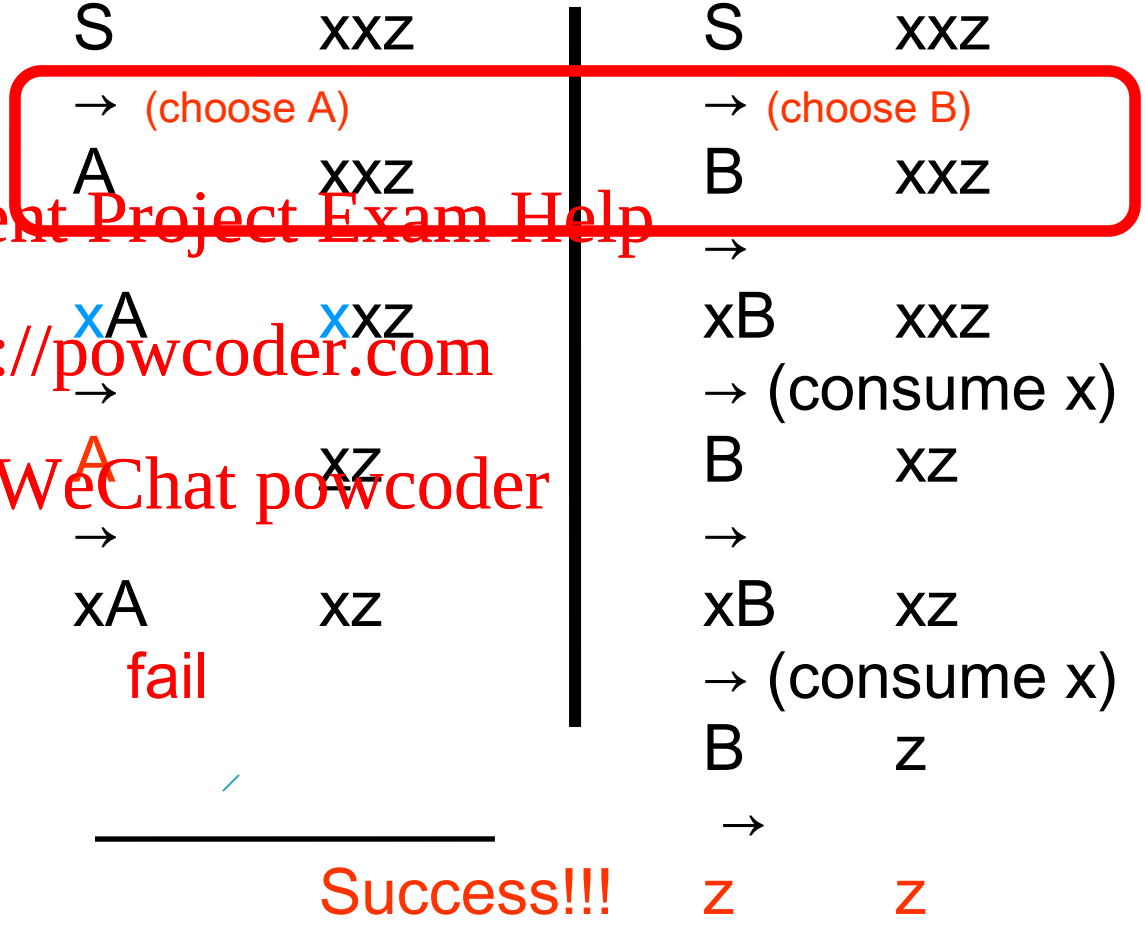
Parsing failed,
but string is in language!!

S xxz
→ (choose A)
A xxz
→
xA xxz
→ (consume x)
A xz
→
xA xz
→ (consume x)
A z

Top-down parsing

S ::= A|B
A ::= xA | y
B ::= xB | z

String to parse: xxz



Lookahead and prediction

- We need to predict which production to choose, preferably by looking only at one symbol
- In this example, we must look at the last symbol, thus required lookahead not even bounded.

S ::= A|B

A ::= xA | y 0 more x's followed by a y or a z

B ::= xB | z

- Need to formulate restrictions on grammars that will allow top down parsing with bounded lookahead.
  - An LL(k) grammar can be parsed by a top-down parser with max k-token lookahead.
  - An LL(*) grammar can be parsed by a top-down parser with unbounded lookahead.

Notational conventions

- It is conventional in general discussions of grammars to use
  - lower case letters near the beginning of the alphabet for terminals
  - lower case letters near the end of the alphabet for strings of terminals
  - upper case letters near the beginning of the alphabet for non-terminals
  - upper case letters near the end of the alphabet for arbitrary symbols
  - Greek letters for arbitrary strings of symbols

FIRST sets

- FIRST(α) is the set of tokens (or ε) that appear as the first symbol in some string generated from α

- FIRST(α) $\equiv$ $\{c : \alpha \rightarrow^* c\ \beta\}$

  - recall α and β are arbitrary strings of symbols while c is a terminal

FIRST sets: Example 1

S ::= AB

A ::= x | y

B ::= z | w

Sentences in language: xz, xw, yz, yw

FIRST(x) = ?????

FIRST sets: Example 1(2)

S ::= AB

A ::= x | y

B ::= z | w

Sentences in language: xz, xw, yz, yw

FIRST(x) = {x}

FIRST(y) = ?????

FIRST sets: Example 1 (3)

S ::= AB

A ::= x | y

B ::= z | w

Sentences in language: xz, xw, yz, yw

FIRST(x) = {x}

FIRST(y) = {y}

FIRST(w) = ?????

FIRST sets: Example 1(4)

S ::= AB

A ::= x | y

B ::= z | w

Sentences in language: xz, xw, yz, yw

FIRST(x) = {x}

FIRST(y) = {y}

FIRST(w) = {w}

FIRST(z} = ?????

FIRST sets: Example 1(5)

S ::= AB

A ::= x | y

B ::= z | w

Sentences in language: xz, xw, yz, yw

FIRST(x) = {x}

FIRST(y) = {y}

FIRST(w) = {w}

FIRST(z) = {z}

FIRST(A) = ????

FIRST sets: Example 1(6)

S ::= AB

A ::= x | y

B ::= z | w

Sentences in language: xz, xw, yz, yw

FIRST(x) = {x}

FIRST(y) = {y}

FIRST(w) = {w}

FIRST(z} = {z}

FIRST(A) =

FIRST(B) = ????

FIRST sets: Example 1(7)

S ::= AB
A ::= x | y
B ::= z | w

Sentences in language: xz, xw, yz, yw

FIRST(x) = {x}

FIRST(y) = {y}
FIRST(w) = {w}

FIRST(z} = {z}
FIRST(A) = FIRST(x) U FIRST(y) = {x,y}
FIRST(B) = FIRST(w) U FIRST(z) = {w,z}
FIRST(S) = ?????

FIRST sets: Example 1(8)

S ::= AB
A ::= x | y
B ::= z | w

Sentences in language: xz, xw, yz, yw

FIRST(x) = {x}
FIRST(y) = {y}
FIRST(w) = {w}
FIRST(z) = {z}

FIRST(A) = FIRST(x) U FIRST(y) = {x,y}
FIRST(B) = FIRST(w) U FIRST(z) = {w,z}
FIRST(S) = FIRST(AB) = FIRST(A) = {x,y}

FIRST sets: Example 2 (we saw this earlier)

S ::= A | B

A ::= xA | y

B ::= xB | z

Sentences in language: y, xy, xxy, …, z, xz, xxz,..

FIRST(xA) = ?????

FIRST sets: Example 2 (2)

S ::= A | B

A ::= xA | y

B ::= xB | z

Sentences in language: y, xy, xxy, …, z, xz, xxz,..

FIRST(xA) = FIRST(x) = {x}

FIRST(y) = ?????

FIRST sets: Example 2 (3)

S ::= A | B

A ::= xA | y

B ::= xB | z

Sentences in language: y, xy, xxy, …z, xz, xxz,..

FIRST(xA) = FIRST(x) = {x}

FIRST(y) = {y}

FIRST(xB) = ?????

FIRST set: Example 2 (4)

S ::= A | B

A ::= xA | y

B ::= xB | z

Sentences in language: y, xy, xxy, …z, xz, xxz,..

FIRST(xA) = FIRST(x) = {x}

FIRST(y) = {y}

FIRST(xB) = FIRST(x) = {x}

FIRST(z) = ?????

FIRST sets: Example 2 (5)

S ::= A | B
A ::= xA | y
B ::= xB | z

Sentences in language: y, xy, xxy, …z, xz, xxz,..

FIRST(xA) = FIRST(x) = {x}
FIRST(y) = {y}
FIRST(xB) = FIRST(x) = {x}
FIRST(z) = {z}
FIRST(A) = ?????

FIRST sets: Example 2 (6)

S ::= A | B

A ::= xA | y

B ::= xB | z

Sentences in language: y, xy, xxy, …z, xz, xxz,..

FIRST(xA) = FIRST(x) = {x}

FIRST(y) = {y}

FIRST(xB) = FIRST(x) = {x}

FIRST(z) = {z}

FIRST(A) = FIRST(xA) U FIRST(y) = {x,y}

FIRST(B) = ?????

FIRST sets: Example 2 (7)

S ::= A | B
A ::= xA | y
B ::= xB | z

Sentences in language: y, xy, xxy, ...z, xz, xxz,..

FIRST(xA) = FIRST(x) = {x}
FIRST(y) = {y}
FIRST(xB) = FIRST(x) = {x}
FIRST(z) = {z}
FIRST(A) = FIRST(xA) U FIRST(y) = {x,y}
FIRST(B) = FIRST(xB) U FIRST(z) = {x,z}
FIRST(S) = ?????

FIRST sets: Example 2 (8)

S ::= A | B
A ::= xA | y
B ::= xB | z

Sentences in language: y, xy, xxy, ..., z, xz, xxz,...

FIRST(xA) = FIRST(x) = {x}
FIRST(y) = {y}
FIRST(xB) = FIRST(x) = {x}
FIRST(z) = {z}
FIRST(A) = FIRST(xA) U FIRST(y) = {x,y}
FIRST(B) = FIRST(xB) U FIRST(z) = {x,z}
FIRST(S) = FIRST(A) U FIRST(B) = {x,y,z}

FIRST set: Example 3

S ::= Ay

A ::= x | ε

Sentences in language: xy, y

FIRST(x) = { x }

FIRST(A) = { x }

FIRST(S) = { x, y }

- Check answer by looking at legal sentences

First requirement for LL(1)

- Recall that we started this discussion of FIRST sets by giving an example where we couldn't predict which production to choose just by looking at the first character.
- To predict the production to use, we need to satisfy:
  - The FIRST set of all productions with the same left hand sides are disjoint.

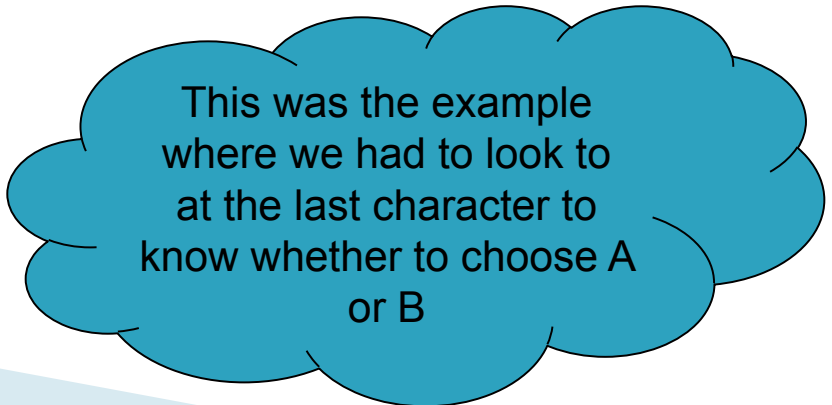
non-LL(1) example, revisited

Example 2

S ::= A
S ::= B
A ::= xA
A ::= y
B ::= xB
B ::= z

(re-written in pure BNF)

Which productions have same left hand side?



non-LL(1) example, revisited

Recall example:

S ::= A
S ::= B
A ::= xA
A ::= y
B ::= xB
B ::= z

FIRST(A) = {x,y}
FIRST(B) = {x,z}

Grammar does not satisfy rule

Another non-LL(1) example

S ::= Ax

A ::= xA | ε

Sentences in the language x, xx, xxx, …

S	x
→ Ax	x
→ xAx	x
→ xxAx	x
→ Ax	ε
FAILURE	

Another non-LL(1) example

S ::= Ax

A ::= xA | ε

Sentences in the language x, xx, xxx, …

S	x
→ Ax	x
→ xAx	x
→ xxAx	x
→ Ax	ε
FAILURE	

Another non-LL(1) example

- Condition on FIRST sets not quite enough

S ::= Ax

A ::= xA | ε

Sentences in the language x, xx, xxx, …

S	x	S	x
→		→	
Ax	x	Ax	x
→		→	
xAx	x	εx	x
→		→	
Ax	ε	ε	ε
FAIL		SUCCESS	

Definition of FOLLOW set

FOLLOW(A) is the set of all terminal symbols that can immediately follow a subsequence derived from A in a sequence derived from the start symbol,S.

$$FOLLOW(A) \equiv \{c : S \rightarrow^{+} \alpha\ A\ c\ \beta\}$$

Example

S ::= AB

A ::= x|y

B ::= z|w

strings in language:
{xz,xw, yz, yw}

FOLLOW(A) = ????

Example

S ::= AB
A ::= x|y
B ::= z|w

strings in language:
{xz,xw, yz, yw}

S
→
AB
→ or →
Az Aw
Thus FOLLOW(A) = FIRST(B) = {w,z}

Example

S ::= AB

A ::= x|y

B ::= z|w

strings in language:

{xz, xw, yz, yw}

FOLLOW(B) = ????

Example

S ::= AB

A ::= x|y

B ::= z|w

strings in language:

{xz, xw, yz, yw}

S

→

AB

FOLLOW(B) = {}

Example

S ::= AB

A ::= x|y

B ::= z|w

strings in language:

{xz, xw, yz, yw}

FOLLOW(x) = ????

Example

S ::= AB
A ::= x|y
B ::= z|w

strings in language:

{xz, xw, yz, yw}

Find x on r.h.s of production. It is rightmost symbol in production. Look at FOLLOW of left side.

Thus

FOLLOW(x) = FOLLOW(A) = {z,w}

Example

S ::= AB

A ::= x|y

B ::= z|w

strings in language {xz,xw, yz, yw}

FOLLOW(A) = FIRST(B) = {z,w}

FOLLOW(B) = {}

FOLLOW(S) = FOLLOW(B) = {}

FOLLOW(x) = FOLLOW(A)

FOLLOW(y) = FOLLOW(A)

FOLLOW(z) = FOLLOW(B)

FOLLOW(w) = FOLLOW(B)

Another Example

S ::= Aw|B

A ::= xA | y

B ::= xB | z

FOLLOW(A) = ???

Another Example

S ::= Aw|B

A ::= xA | y

B ::= xB | z

FOLLOW(A) = FIRST(w) = {w}

Another Example

S ::= Aw|B

A ::= xA | y

B ::= xB | z

FOLLOW(A) = FIRST(w) = {w}

Another Example

S ::= Aw|B

A ::= xA | y

B ::= xB | z

FOLLOW(x) = ?!?!?

Another Example

S ::= Aw|B

A ::= xA | y

B ::= xB | z

Find all occurrences in right hand sides of productions. x is followed by A in one and B in another, thus

FOLLOW(x) = FIRST(A) U FIRST(B) = {x,y,z}

summary

S ::= Aw|B
A ::= xA | y
B ::= xB | z

FOLLOW(A) = FIRST(w) = {w}
FOLLOW(B) = {}
FOLLOW(S) = {}
FOLLOW(x) = FIRST(A) U FIRST(B) = {x,y,z}
FOLLOW(y) = FOLLOW(A) = {w}
FOLLOW(z) = FOLLOW(B) = {}

Still another example

S ::= Ay Strings = {xy, y}

A ::= x | ε

FOLLOW(A) = {y}

FOLLOW(x) = FOLLOW(A) = {y}

FOLLOW(y) = {}

FOLLOW(S) = {}

Recall this example that gave us trouble parsing

S ::= Ax

A ::= xA | ε

Sentences in the language x, xx, xxx, …

S	x		S	x	
→			→		
Ax	x		Ax	x	
→			→		
xAx	x		εx	x	
→			→		
Ax	ε		ε	ε	
FAIL			SUCCESS		

S ::= Ax

A ::= x | ε

FIRST(A) = { x}

FOLLOW(S) = { ε }

FOLLOW(A) = FIRST(x) = { x }

FOLLOW(x) = FOLLOW(A) = { x }

Additional condition:

for all non-terminals A:

if A →* ε, then FIRST(A) ∩ FOLLOW(A) = { }

Predict sets

Combine ideas to get the predict set of a production

Define

EPS(α) = (α ->* ε) (this is either true for false)

PREDICT(A ::= α) = FIRST(α) ∪
(if EPS(α) then FOLLOW(A) else {})

- The predict set tells us which production to choose.
  - If we see non-terminal a, and a in PREDICT(A ::= α), then choose this production
  - For the choice to be unambiguous, the predict sets with same left side should be unique.

LL(1) rule

The predict sets of all productions with the same left side are disjoint.

- Necessary and sufficient for a grammar to be LL(1)

- LL(1) means

  - can be parsed left-to-right (first L)
  - leftmost derivation (second L)
  - one symbol look ahead (the 1)

- We use predict sets for
  - determining whether a grammar is LL(1)
  - constructing the parser

- See Scott for an algorithm to mechanically compute predict sets.
  - Parser generators need to do this

- Remarks

  LL(k) grammars can be parsed with k symbols lookahead

  LL grammars are parsed with top-down parsers

  LR grammars are parsed with bottom-up parsers.

  There are other classifications, if you are interested, see the Scott CS supplement