Parsing

CIS400 (Compiler Construction)

Kris Micinski, Fall 2021

Parsing Overview

- Parsing translates a raw character stream into an abstract syntax tree (in Racket we use only S-expressions, which can be used to build trees)
- The first phase of parsing (often a separate phase) is called *lexical analysis* (lexing), which breaks a character stream into a *token* stream
- These tokens are then used as the building blocks (terminals) of a *context-free grammar* for some language
- A parser recognizes (and constructs an AST for) some language for us to then manipulate

First, a digression on lexing

Let's assume the **get-token** function will give me the next token

Lexing

- Lexical analysis breaks the character-based input stream into a set of **tokens** typically specified via a set of **regular expressions**
  - E.g., the language of a `1` followed by one or more `0`s
    - `100*` (eqv. `10+`) matches `10`, `100`, but not `1` or `110`
  - Regular expressions classify the **regular languages**
    - Formal class of languages possible to recognize using only a **single state of memory** or (equivalently) a **finite state automaton**

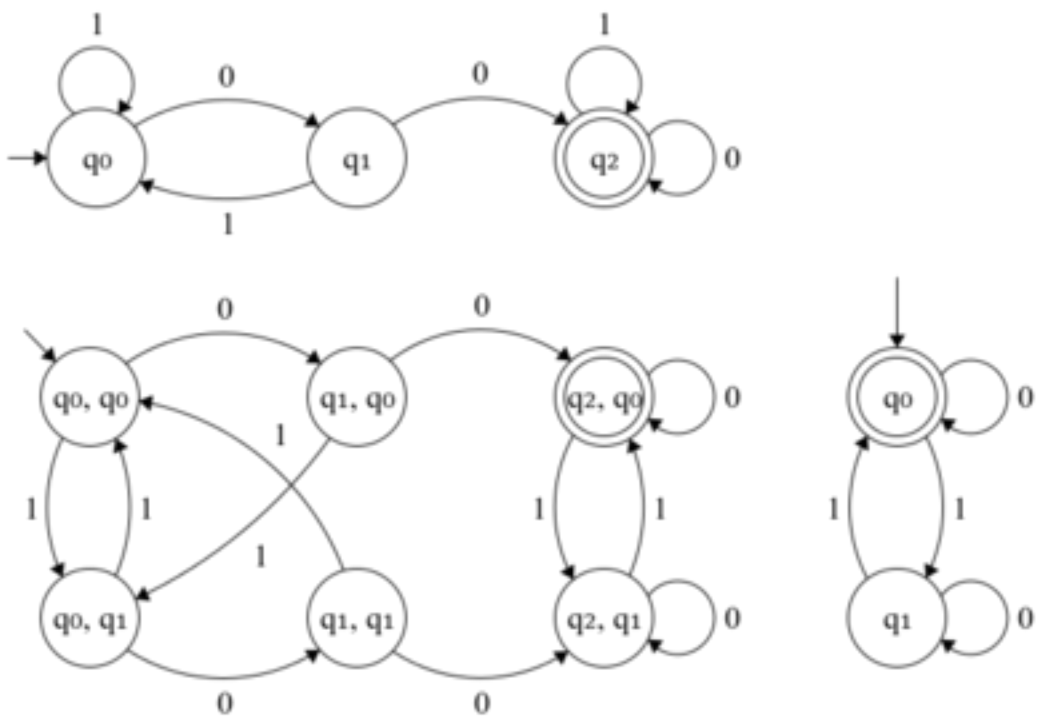
Regular expressions

`100*    (1|0)*1`

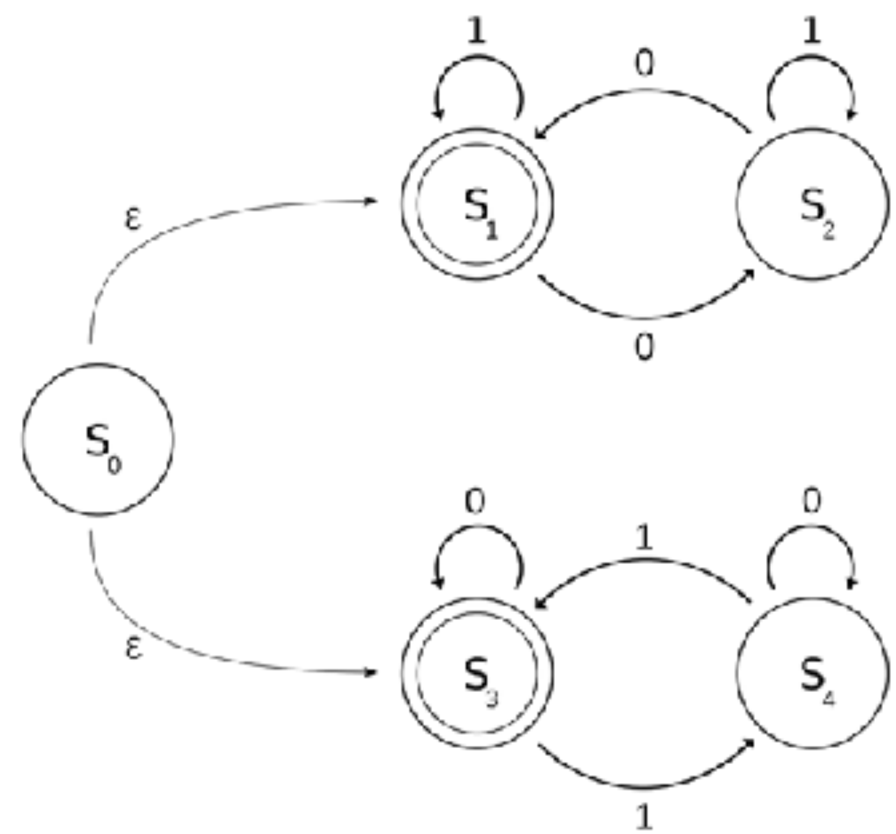
`(10)*(01)*`

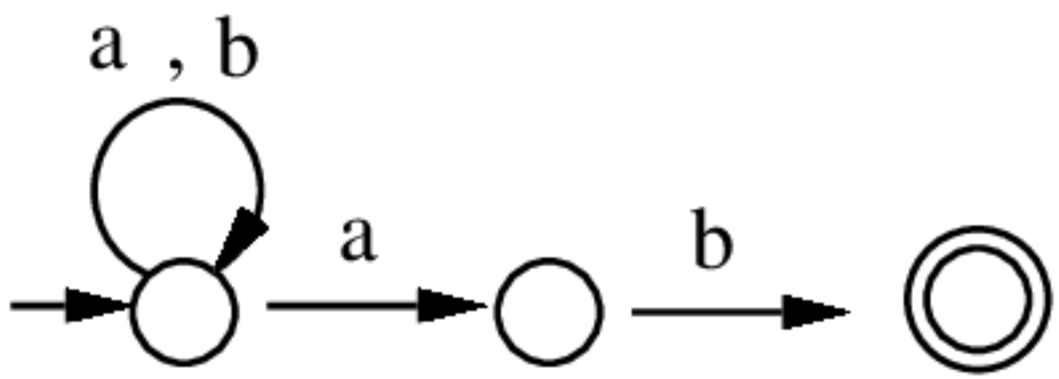
Big lesson from CS theory:
these three are **all equivalent!**

Deterministic Finite Automata



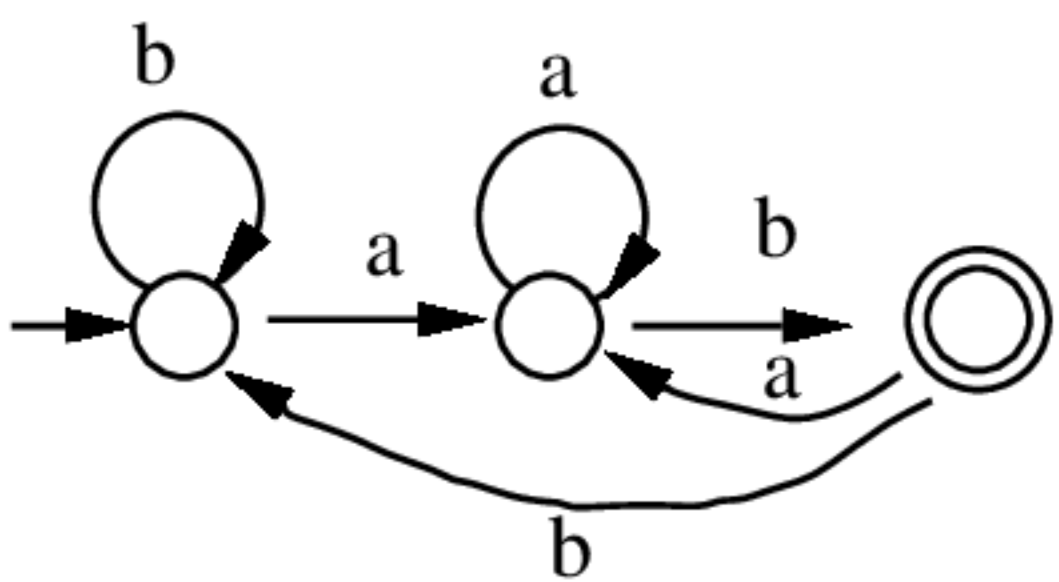
Nondeterministic Finite Automata





NFA

for $\boldsymbol{M}[\boldsymbol{p}]$



DFA

for $\boldsymbol{M}[\boldsymbol{p}]$

Lexing continued…

- Once you define the set of tokens, you give these to the lexing library and the lexing library then translates the (character-based) input stream into a **token stream**
- Lexing very interesting from a CS theory perspective
  - Very boring from a programming perspective—just use off-the-shelf tools / libraries
- But the tokens the lexer produces become the **terminal** symbols of our context-free grammars

Racket’s parser-tools/lex

```
(define lex
 (lexer
    ; skip spaces:
    [#\space     (lex input-port)]
    ; skip newline:
    [#\newline   (lex input-port)]

    [#\+         'plus]
    [#\-         'minus]
    [#\*         'times]
    [#\/         'div]

    [(:: (:? #\-) (:+ (char-range #\0 #\9)))
      (string->number lexeme)]
    ; an actual character:
    [any-char    (string-ref lexeme 0)]))
```

Context Free Grammars

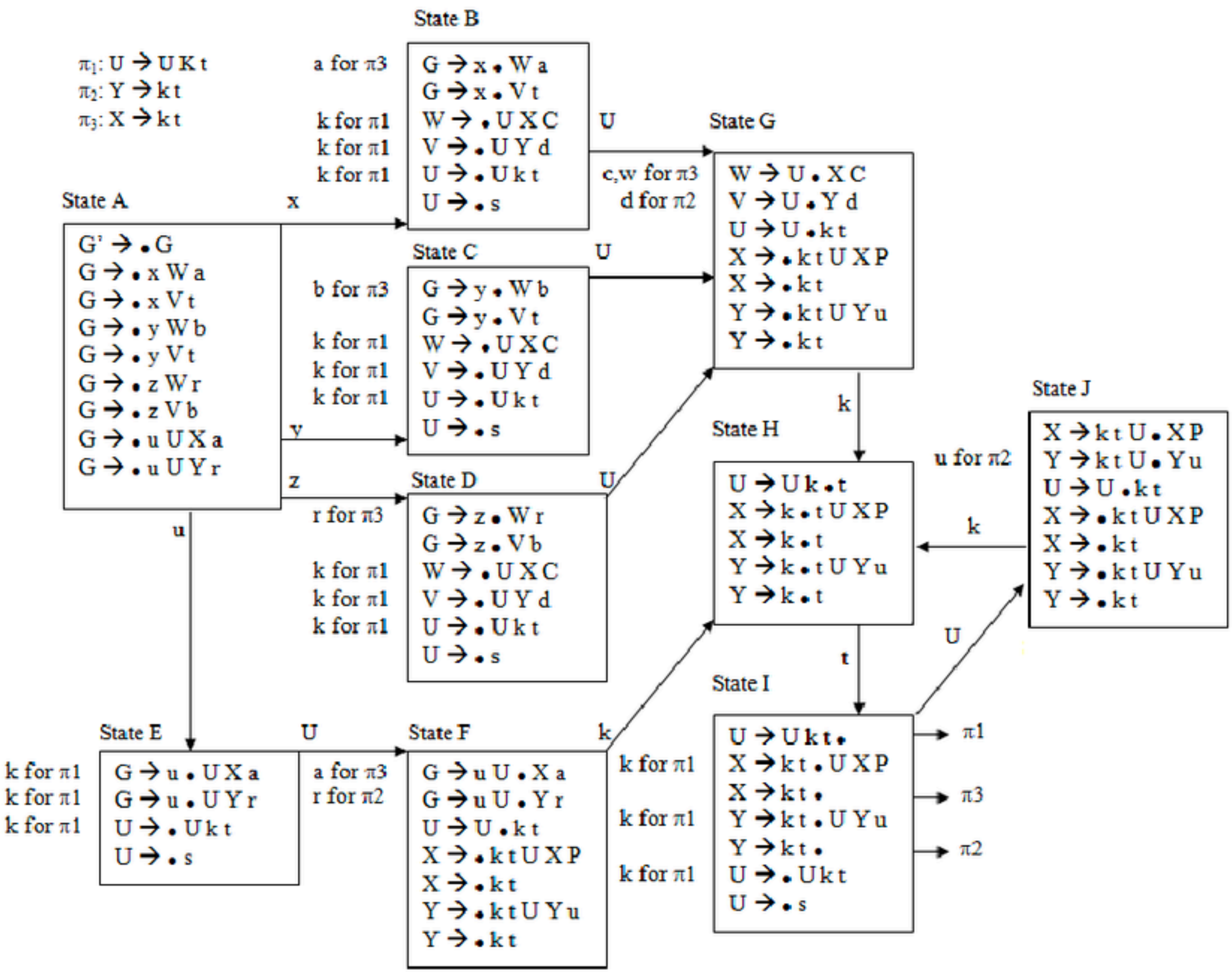
- Computability theory establishes a hierarchy of computational models via classes of **languages**
- **Context-free** grammars (CFGs) characterize context-free languages
  - ***More expressive*** than regular languages
  - For example, the language of balanced parentheses ({}, {{}}, {{{}}}, …) is context-free but **not regular**
    - There is **no** regular expression that characterizes balanced parens!

CFG structure

- Consists of a set of **terminals** (i.e., tokens)
- A set of **nonterminals** (inferred or derived tokens)
- A set of **production rules** establishing how to generate grammar
  - ```
    addexpr ::= mulexpr
             |  mulexpr + addexpr
    mulexpr ::= num
             |  mulexpr * mulexpr
    ```
- Using this, we then define when a grammar **accepts** an input string
    ```

CS theory stuff for CFGs…

- REs are realized on machines as **finite state machines** which can be executed extremely efficiently via lookup tables
- CFGs use a similar implementation, but also add a **stack** of tokens.
- This forms a **pushdown automata** (PDA), a finite state machine that also gets to use a stack to push on / off
- This set of languages is **still** limited in its ability to compute, as it must process in a stack-oriented fashion with a finite set of states
- **Turing machines** TMs generalize PDAs, allow **more** languages!
    - $\{0^i\ 1^j\ 2^k \mid 0 <= i <= j <= k\}$
    - Not CFG, proved via pumping lemma



π1: U → U K t
π2: Y → k t
π3: X → k t
State A
G' → • G
G → • x W a
G → • x V t
G → • y W b
G → • y V t
G → • z W r
G → • z V b
G → • u U X a
G → • u U Y r
State B
G → x • W a
G → x • V t
W → • U X C
V → • U Y d
U → • U k t
U → • s
a for π3
k for π1
k for π1
k for π1
State C
G → y • W b
G → y • V t
W → • U X C
V → • U Y d
U → • U k t
U → • s
b for π3
k for π1
k for π1
k for π1
State D
G → z • W r
G → z • V b
W → • U X C
V → • U Y d
U → • U k t
U → • s
r for π3
k for π1
k for π1
k for π1
State E
G → u • U X a
G → u • U Y r
U → • U k t
U → • s
k for π1
k for π1
k for π1
State F
G → u U • X a
G → u U • Y r
U → U • k t
X → • k t U X P
X → • k t
Y → • k t U Y u
Y → • k t
a for π3
r for π2
k for π1
k for π1
k for π1
State G
W → U • X C
V → U • Y d
U → U • k t
X → • k t U X P
X → • k t
Y → • k t U Y u
Y → • k t
c,w for π3
d for π2
State H
U → U k • t
X → k • t U X P
X → k • t
Y → k • t U Y u
Y → k • t
State I
U → U k t •
X → k t • U X P
X → k t •
Y → k t • U Y u
Y → k t •
U → • U k t
U → • s
π1
π3
π2
State J
X → k t U • X P
Y → k t U • Y u
U → U • k t
X → • k t U X P
X → • k t
Y → • k t U Y u
Y → • k t
u for π2
x
y
z
u
U
U
U
U
U
U
k
k
k
t

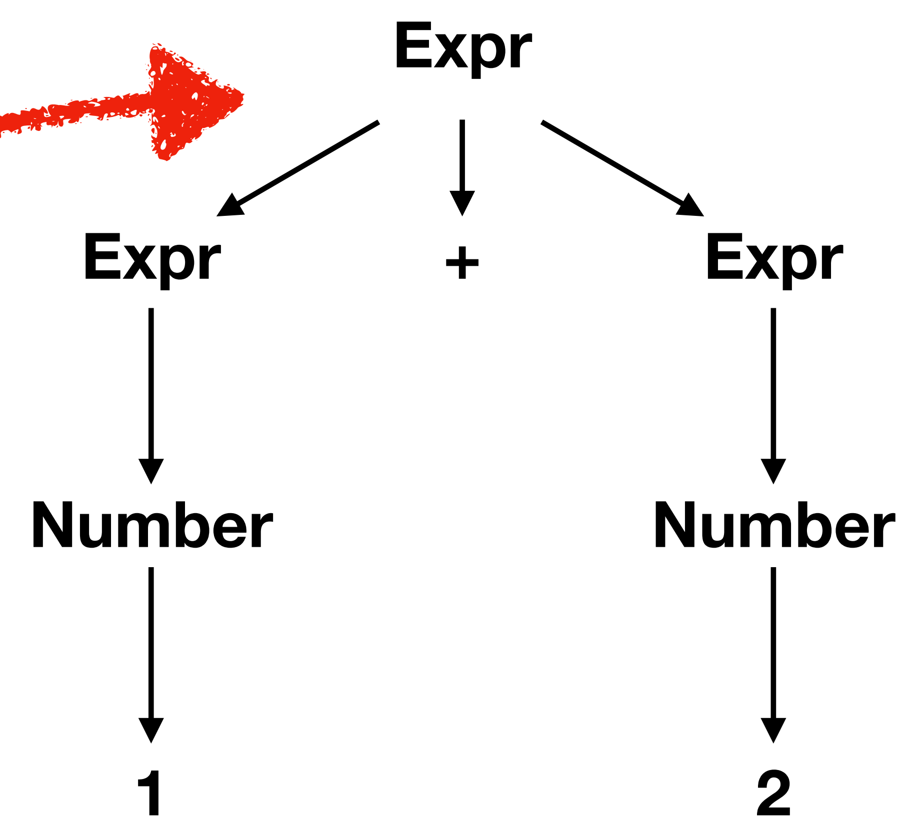
```
Expr -> number
Expr -> Expr + Expr
Expr -> Expr * Expr
```

1 + 2 * 3

```
Expr
-> Expr + Expr
-> Expr + Expr * Expr
-> number + Expr * Expr
-> number + number * Expr
-> number + number * number
```

```
Expr
-> Expr * Expr
-> Expr + Expr * Expr
-> number + Expr * Expr
-> number + number * Expr
-> number + number * number
```

```
Expr
-> Expr + Expr
-> number + Expr
-> number + number
-> 1 + number
-> 1 + 2
```



This parse tree is a **hierarchical representation** of the data

A **parser** is a program that automatically generates a parse tree

A parser will generate an **abstract syntax tree** for the language

Exercise: draw the parse trees for the following derivations

```
Expr
-> Expr + Expr
-> Expr + Expr * Expr
-> number + Expr * Expr
-> number + number * Expr
-> number + number * number
```

```
Expr
-> Expr * Expr
-> Expr + Expr * Expr
-> number + Expr * Expr
-> number + number * Expr
-> number + number * number
```

BNF

(Bakus-Naur Form)

```
<Expr> ::= <number>
<Expr> ::= <Expr> + <Expr>
<Expr> ::= <Expr> * <Expr>
```

Slightly different form for writing CFGs, superficially different

(BNF renders nicely in ASCII, but no huge differences)

I write colloquially in some mix of BNF and more math style

Two kinds of derivations

Leftmost derivation: The leftmost nonterminal is expanded first at each step

Rightmost derivation: The rightmost nonterminal is expanded first at each step

```
G -> GG
G -> a
```

Draw the **leftmost derivation** for…

aaa

Draw the **rightmost derivation** for…

aaa

```
G -> G + G
G -> G / G
G -> number
```

Draw a leftmost derivation for…

```
1 / 2 / 3
```

Now draw *another* leftmost derivation

Draw the parse trees for each derivation

What does each parse tree mean?

A grammar is **ambiguous** if there is a string with **more than one** leftmost derivation

(Equiv: has more than one parse tree)

Generally, we’re going to want our grammar to be **unambiguous**

```
G -> G + G
G -> G / G
G -> number
```

There's another problem with this grammar

We need to tackle **ambiguity**

Idea: introduce extra nonterminals that force you to get left-associativity

```
Add -> Add + Mul | Mul
Mul -> Mul / Term | Term
Term -> number
```

Write derivation for 5 / 3 / 1

Draw the parse tree for 5 / 3 / 1

```
Add -> Add + Mul | Mul
Mul -> Mul / Term | Term
Term -> number
```

This grammar is **left recursive**

```
Add -> Add + Mul | Mul
Mul -> Mul / Term | Term
Term -> number
```

A grammar is left-recursive if any nonterminal A has a production of the form A -> A…

```
Add -> Add + Mul | Mul
Mul -> Mul / Term | Term
Term -> number
```

This will turn out to be bad for one class of parsing algorithms

Assume current token is `curtok`

`(accept c)` matches character c

```
(define curtok (next-tok))

(define (accept c)
  (if (not (equal? curtok c))
      (raise 'unexpected-token)
      (begin
        (printf "Accepting ~a\n" c)
        (set! curtok (next-tok)))))
```

L eft to right

L eft derivation

1 token of lookahead

Let’s say I want to parse the following grammar

S -> aSa | bb

First, a few questions

$$S \to aSa \mid bb$$

Is this grammar ambiguous?

If I were matching the string **bb,** what would my derivation look like?

If I were matching the string **abba**, what would my derivation look like?

First, a few questions

$$S \rightarrow aSa \mid bb$$

Key idea: if I look at the next input, at most one of these productions can "fire"

If I see an **a** I **know** that I **must** use the first production

If I see a b, I know I must be in second production

This is called a **predictive** parser. It uses lookahead to determine which production to choose

(My friend Tom points out that **predictive** is a dumb name because it is really “determining”, no guess)

In this class, we'll restrict ourselves to grammars that require only **one** character of lookahead

Generalizing to k characters is straightforward

I need two characters of lookahead

S -> aaS | abS | c

I need three characters of lookahead

S -> aaaS | aabS | c

I need four characters of lookahead

S -> aaaaS | aaabS | c

...

Slight transformation..

```
S  ->  A  |  B
A  ->  aSa
B  ->  bb
```

Slight transformation..

```
S -> A | B
A -> aSa
B -> bb
```

Now, I write out **one function** to parse **each** nonterminal

```
S -> A | B
A -> aSa
B -> bb
```

Intuition: when I see **a**, I call `parse-A`

when I see **b**, I call `parse-B`

```
(define (parse-A)
  (match curtok
    [#\a
     (begin
       (accept #\a)
       (parse-A)
       (accept #\a))]
    [#\b (parse-B)]))
```

```
(define (parse-B)
  (begin
    (accept #\b)
    (accept #\b)))
```

Three parsing-related pieces of trivia

FIRST(A)

FIRST(A) is the **set** of terminals that could occur **first** when I recognize A

NULLABLE

Is the set productions which could generate ε

FOLLOW(A)

FOLLOW(A) is the set of terminals that appear immediately to the right of A in some form

Why learn these?

A: They help your intuition for building parsers
(as we’ll see)

What is FIRST for each nonterminal

S -> A | B
A -> aAa
B -> bb

What is NULLABLE for the grammar

What is FOLLOW for each nonterminal

More practice…

E → TE'

E' → +TE'

E' → ε

T → FT'

T' → *FT'

T' → ε

F → (E)

F → id

What is FIRST for each nonterminal

What is NULLABLE for the grammar

What is FOLLOW for each nonterminal

We use the **FIRST** set to help us design our recursive-descent parser!

LL(1)

A grammar is LL(1) if we only have to look at the **next** token to decide which production will match!

I.e., if S -> A | B, FIRST(A) $\cap$ FIRST(B) must be empty

Recursive-descent is called **top-down** parsing because you build a parse tree from the root down to the leaves

There are also **bottom-up** parsers,
which produce the rightmost derivation

Won't talk about them, in general they're impossibly-hard
to write / understand by hand without automation

UNIX Programming Tools

2nd Edition

lex & yacc

O'REILLY®

John R. Levine, Tony Mason & Doug Brown

Most people use a parsing library like lex & yacc (or some equivalent for modern languages, e.g., Racket's parser-tools/lex)

Recursive-descent is easy to implement, but may require restructuring the grammar to achieve LL(k)

More practice with parsers

This one is more tricky!!

```
Plus -> num MoreNums
MoreNums -> + num MoreNums | ε
```

How would you do it?

(**Hint:** Think about NULLABLE)

Code up collectively….

```
(define (parse-Plus)
  (begin
    (parse-num)
    (parse-MorePlus)))

(define (parse-MorePlus)
  (match curtok
    ['plus
     (begin
       (accept 'plus)
       (parse-num)
       (parse-MorePlus))]
    ['eof (void)]))
```

Key rule: At each step of the way, if I see some token next, what rule production **must** I choose

Now yet another….

This will use the intuition from FOLLOW

```
Add -> Term MoreTerms
MoreTerms -> + Term MoreTerms
MoreTerms -> ε
Term -> num MoreNums
MoreNums -> * num MoreNums | ε
```

Consider how we would implement `MoreTerms`

```
Add -> Term MoreTerms
MoreTerms -> + Term MoreTerms
MoreTerms -> ε
Term -> num MoreNums
MoreNums -> * num MoreNums | ε
```

If you're at the beginning of `MoreTerms` you **have** to see a +

```
Add -> Term MoreTerms
MoreTerms -> + Term MoreTerms
MoreTerms -> ε
Term -> num MoreNums
MoreNums -> * num MoreNums | ε
```

If you've just seen a + you have to see FIRST(Term)

```
Add -> Term MoreTerms
MoreTerms -> + Term MoreTerms
MoreTerms -> ε
Term -> num MoreNums
MoreNums -> * num MoreNums | ε
```

After Term you recognize something in FOLLOW(Term)

```
Add -> Term MoreTerms
MoreTerms -> + Term MoreTerms
MoreTerms -> ε
Term -> num MoreNums
MoreNums -> * num MoreNums | ε
```

Because MoreTerms is NULLABLE, have to account for null

```
Add -> Term MoreTerms
MoreTerms -> + Term MoreTerms
MoreTerms -> ε
Term -> num MoreNums
MoreNums -> * num MoreNums | ε
```

Code up collectively….

Let’s say I want to generate an AST

Model my AST…

```
(struct add (left right) #:transparent)
(struct times (left right) #:transparent)
```

More Recursive-descent practice…

Write recursive-descent parsers for the following….

A grammar for S-Expressions

Parsing mini-Racket / Scheme

```
datum ::= number
        | string
        | identifier
        | ‘SExpr
SExpr ::= (SExprs)
        | datum
SExprs ::= SExpr SExprs
         | ε
```

```
S -> a C H | b H C
H -> b H | d
C -> e C | f C
```

```
E -> A
E -> L
A -> n
A -> i
L -> ( S )
S -> E S'
S' -> , S
S' -> ε
```

These have all been LL(1) grammars

(Many grammars are **not**)

But you can often transform them to LL(1)

What about this grammar?

```
E -> E - T | T
T -> number
```

This grammar is **left recursive**

```
E -> E - T | T
T -> number
```

What happens if we try to write recursive-descent parser?

This grammar is **left recursive**

```
E -> E - T | T
T -> number
```

We really **want** this grammar, because it corresponds to the **correct** notion of associativity

```
E -> E - T | T
T -> number
```

5 - 3 - 1

Infinite loop!

```
E -> E - T | T
T -> number
```

$$5 - 3 - 1$$

A recursive descent parser will first call `parse-E`

And then crash

```
E -> E - T | T
T -> number
```

```
5 - 3 - 1
```

Draw the **rightmost derivation** for this string

If we could only have the **rightmost** derivation, our problem would be solved

The problem is, a recursive-descent parser needs to look at the **next input immediately**

Recursive descent parsers work by looking at the next token and making a decision / prediction

Rightmost derivations require us to delay making choices about the input until later

As humans, **we** naturally guess which derivation to use (for small examples)

Thus, LL(k) parsers cannot generate rightmost derivations :(

We can **remove left recursion**

```
E -> E - T | T
T -> number
```

Factor!

```
E  -> T E'
E' -> - T E'
E' -> ε
```

In general, if we have

```
A -> Aα | bB
```

Rewrite to…

```
A -> bB A'
A' -> α A' | ε
```

Generalizes even further

https://en.wikipedia.org/wiki/LL_parser#Left_Factoring

Unfortunately, this still produces the wrong grouping of -

```
E  -> T E'
E' -> - T E'
E' -> ε
```

```
E -> T E'
  -> T - T E'
  -> T - T - T E'
  -> T - T - T
```

So how do we get left associativity?

If writing recursive-descent parser by hand, can hack implementation to swap in the right thing..

If you want to get **rightmost** derivation, you need to use an LR parser

```
// example from yacc
input:    /* empty */
        | input line
;

line:     '\n'
        | exp '\n'  { printf ("\t%.10g\n", $1); }
;

exp:      NUM                { $$ = $1;         }
        | exp exp '+'        { $$ = $1 + $2;    }
        | exp exp '-'        { $$ = $1 - $2;    }
        | exp exp '*'        { $$ = $1 * $2;    }
        | exp exp '/'        { $$ = $1 / $2;    }
      /* Exponentiation */
        | exp exp '^'        { $$ = pow ($1, $2); }
      /* Unary minus    */
        | exp 'n'            { $$ = -$1;        }
;
```

Parting Thoughts

- Writing parsers is notoriously prone to errors
  - OTOH most standard tools for writing parsers are really bad (e.g., shift-reduce and reduce-reduce conflicts from yacc)/hard to use
- If you must write a parser, try to only write a simple LL(k) style parser and implement via recursive descent
  - Crucial insight: call-return matching of the program's stack mirrors the natural parsing structure inherent to the grammar
- But mostly, **avoid writing parsers.** Stick to standard input formats (e.g., JSON/S-expressions/…) when possible
  - Able to use standard, well-tested parsers! Avoid painful and unnecessary debugging!