CS 152

Programming Paradigms

Parsers: Recursive Descent

Announcement

- Office hours on Thursday, October 8 will start at 3:30 PM instead of 3 PM.

Today

- EBNF and Syntax Diagrams
- Recursive descent parsers
- Made Up Functional Language MUFL 1.0 (Homework 6)

Course Learning Outcomes

5. Read and produce context-free grammars
6. Write recursive-descent parsers for simple languages, by hand or with a parser generator.
8. Write interpreters for simple languages that involve arithmetic expressions, bindings of values to names, and function calls.

Extended Backus-Naur form

Two new metasymbols:

- { } Curly braces
- [ ] Square brackets

Left Recursive Productions

<expr> → <expr> PLUS <term> | <term>

<expr>

⇒ <expr> PLUS <term>

⇒ <expr> PLUS <term> PLUS <term>

⇒ <expr> PLUS <term> PLUS <term> PLUS <term>

⇒ ...

⇒ <term> PLUS ... PLUS <term>

Arbitrary number of terms separated by PLUS.

Notation: <expr> → <term> { PLUS <term>}

{ }: 0 or more

Left Recursive Productions

BNF: <expr> → <expr> PLUS <term> | <term>

EBNF: <term> { PLUS <term>}

- We are essentially replacing left recursion with a loop
- Assumption: any operator involved in a curly bracket repetition is left-associative
- Useful in writing recursive descent parsers

Optional Parts

<if_stmt> → IF <test> : <suite> |

IF <test> : <suite> ELSE : <suite>

The 'else' branch is optional.

Notation: <if_stmt> → IF <test> : <suite> [ELSE : <suite>]

[ ]: 0 or 1 - optional

Right Recursive Productions

<expr> → <term> PLUS <expr> | <term>

EBNF Notation:

<expr> → <term> [PLUS <expr>]

Arithmetic Expressions BNF

<expr> → <expr> PLUS <term> | <expr> MINUS <term> | <term>
<term> → <term> TIMES <factor> | <factor>
<factor> → LPAREN <expr> RPAREN | <number>
<number> → INT | FLOAT

Combine the tokens PLUS and MINUS into one token ADD_OP: includes the operators + and –

Arithmetic Expressions BNF

<expr> → <expr> PLUS <term> | <expr> MINUS <term> | <term>
<term> → <term> TIMES <factor> | <factor>
<factor> → LPAREN <expr> RPAREN | <number>
<number> → INT | FLOAT

<expr> → <expr> ADD_OP <term> | <term>
<term> → <term> TIMES <factor> | <factor>
<factor> → LPAREN <expr> RPAREN | <number>
<number> → INT | FLOAT

iClicker: EBNF?

<expr> → <expr> ADD_OP <term> | <term>

A. <expr> → <expr> {ADD_OP <term>}
B. <expr> → <expr> [ADD_OP <term>]
C. <expr> → <term> {ADD_OP <term>}
D. <expr> → <term> [ADD_OP <term>]
E. None of the above

iClicker: EBNF?

<term> → <term> TIMES <factor> | <factor>

A. <term> → <term> {TIMES <factor>}
B. <term> → <term> [TIMES <factor>]
C. <term> → <factor> [TIMES <factor>]
D. <term> → <factor> {TIMES <factor>}
E. None of the above

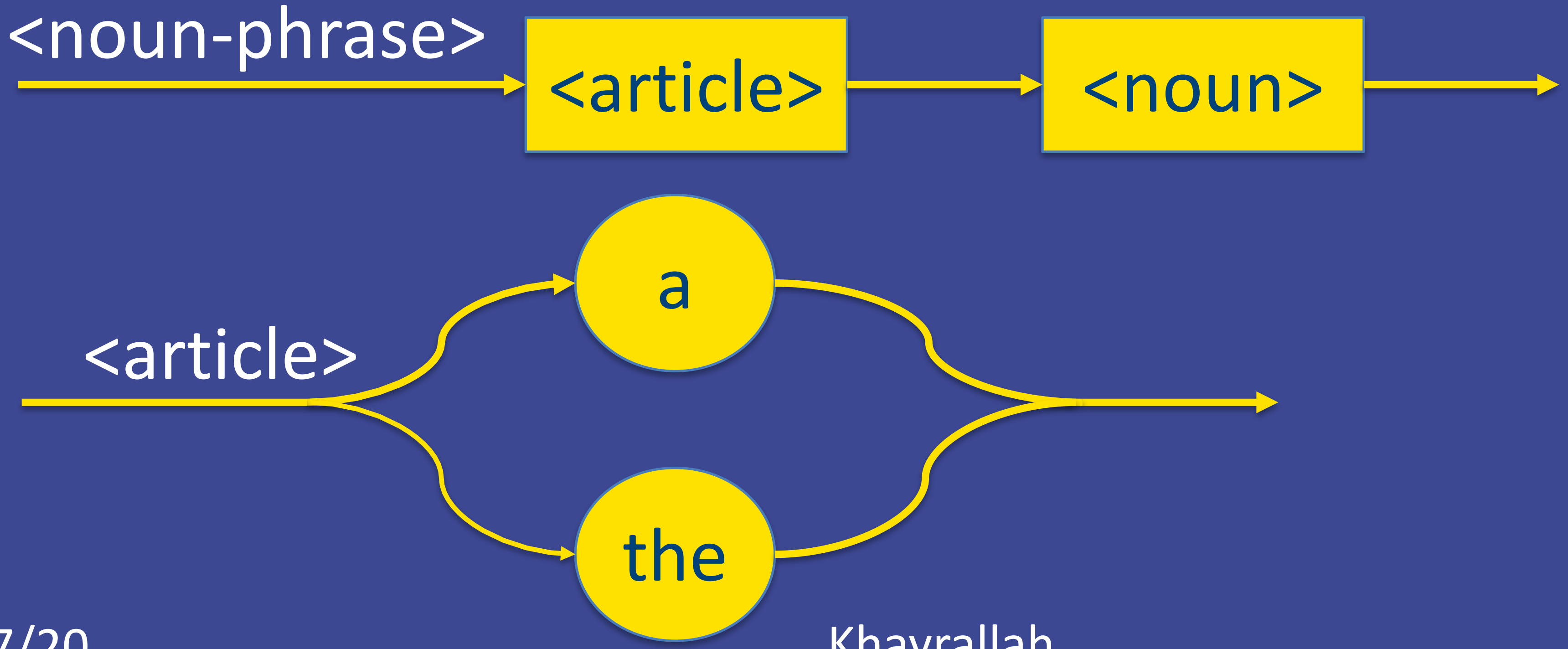
Arithmetic Expressions EBNF

<expr> → <term> {ADD_OP <term>}
<term> → <factor> {TIMES <factor>}
<factor> → LPAREN <expr> RPAREN | <number>
<number> → INT | FLOAT

Syntax Diagrams

<noun-phrase> → <article> <noun>

<article> → a | the



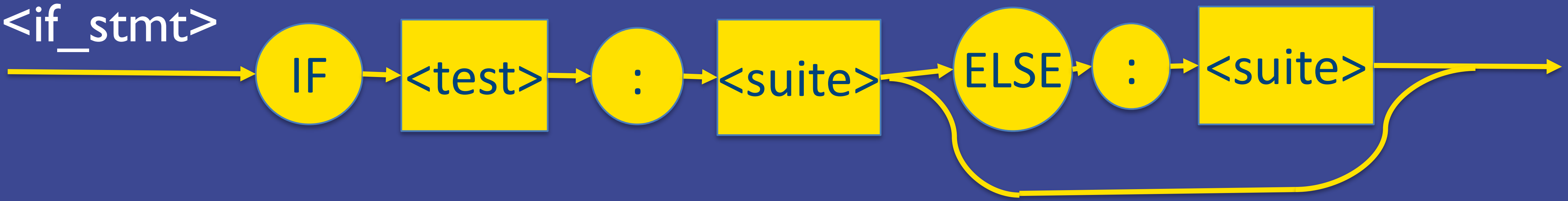
Syntax diagrams

- Represent the sequence of terminals and non-terminals encountered in the right-hand side of the rule
- Use circles or ovals for terminals, and squares or rectangles for non-terminals
- Connect them with lines and arrows indicating appropriate sequencing
- Can condense several rules into one diagram
- Use loops to indicate repetition
- Always based on EBNF

Syntax Diagrams

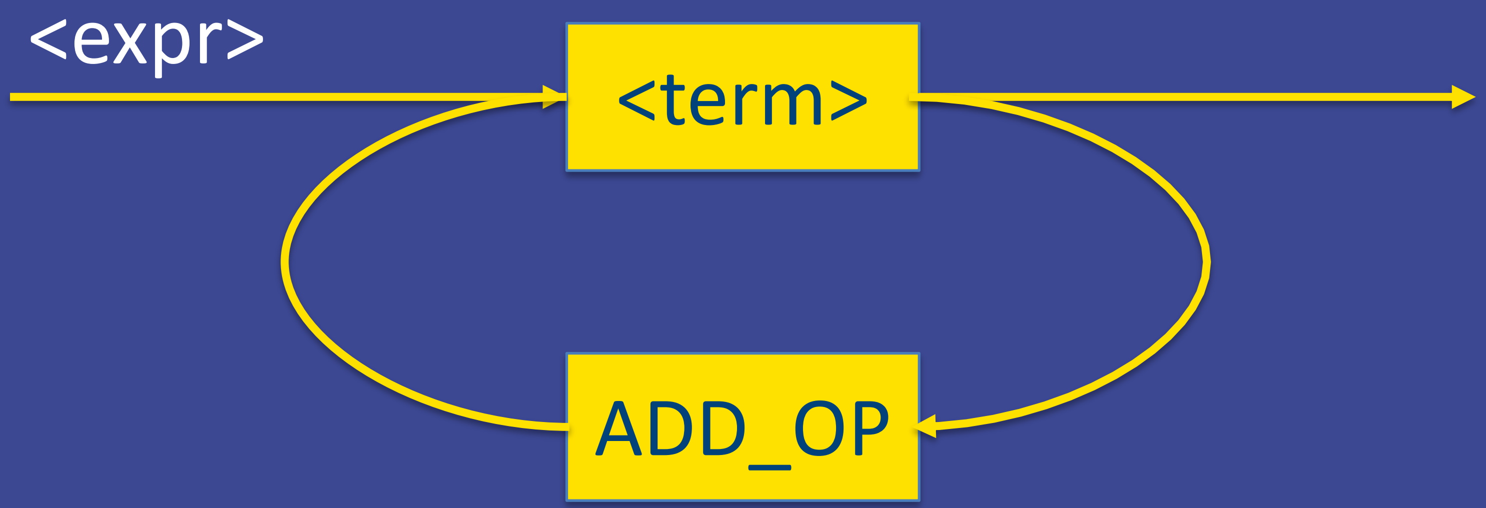
<if_stmt> → IF <test> ':' <suite> [ELSE ':' <suite>]

<if_stmt>



Syntax Diagrams

<expr> → <term> {ADD_OP <term>}

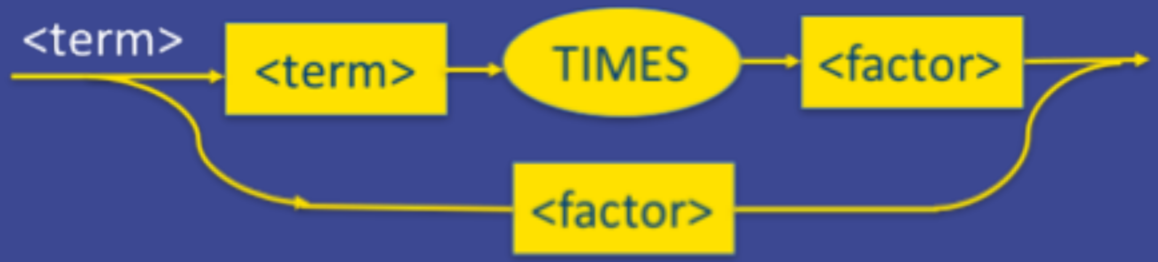


iClicker: Syntax Diagram?

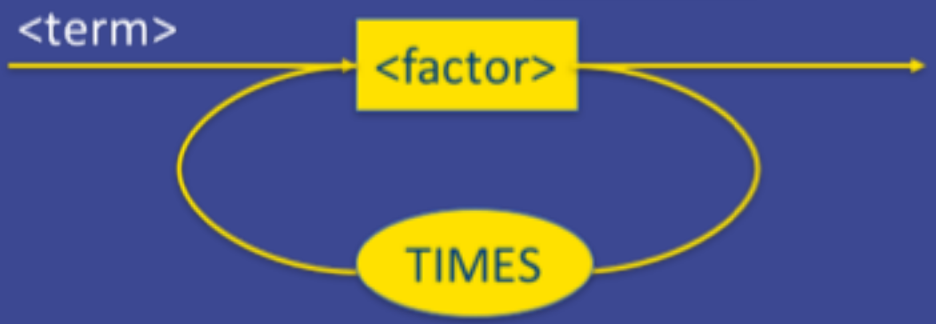
BNF: <term> → <term> TIMES <factor> | <factor>

EBNF: <term> → <factor> {TIMES <factor>}

A.



B.

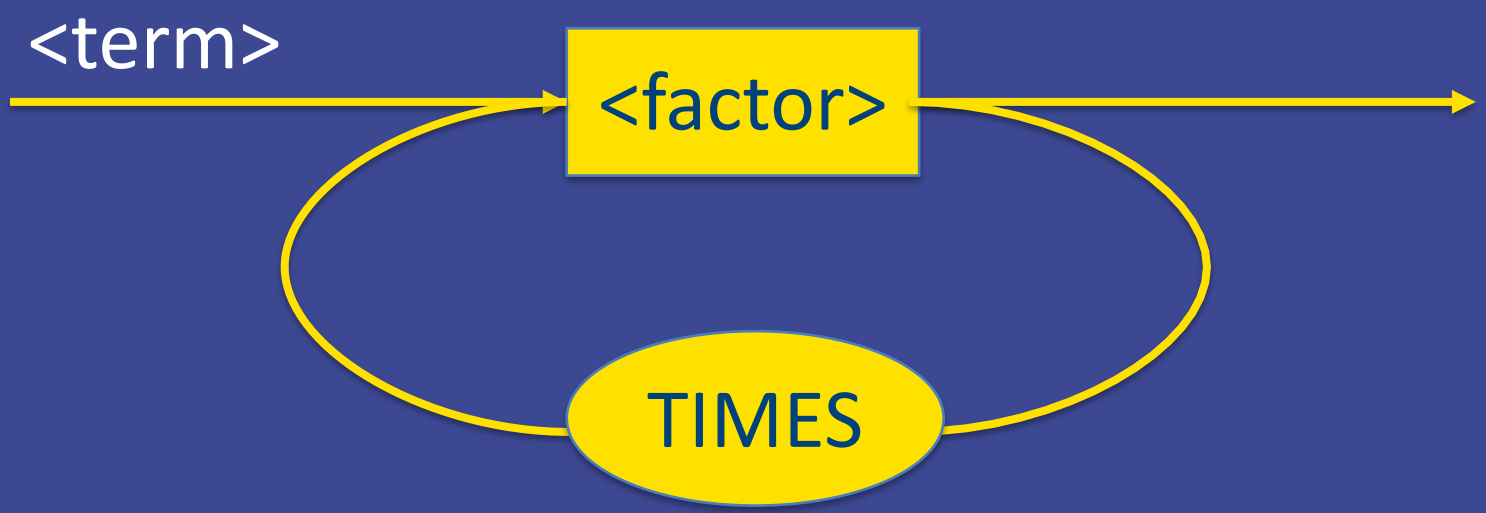


C. Both diagrams are valid

Syntax Diagrams

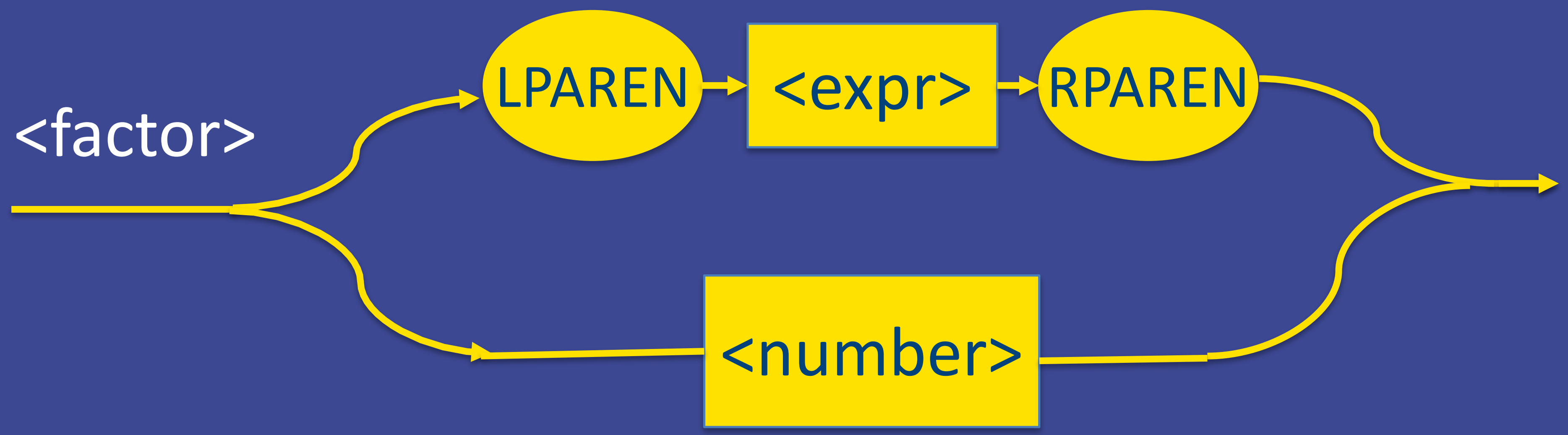
BNF: <term> → <term> TIMES <factor> | <factor>

EBNF: <term> → <factor> {TIMES <factor>}



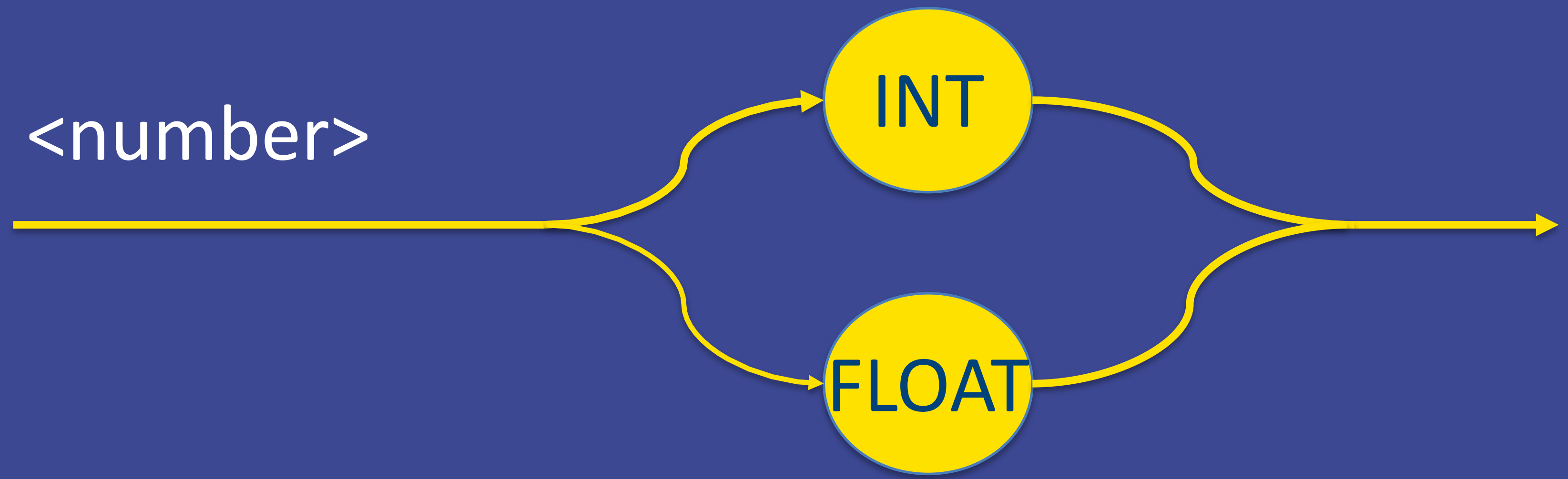
Syntax Diagrams

<factor> → LPAREN <expr> RPAREN | <number>



Syntax Diagrams

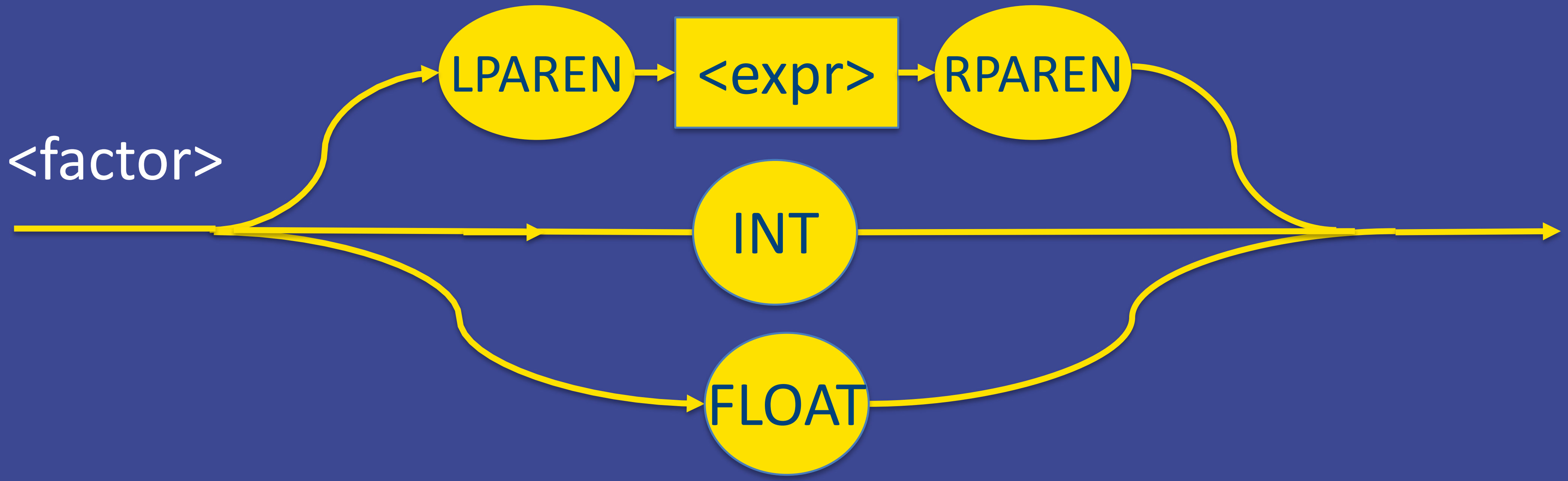
<number> → INT | FLOAT



Syntax Diagrams

<factor> → LPAREN <expr> RPAREN | <number>

<number> → INT | FLOAT



- Can condense several rules into one diagram

EBNF for Prefix Calculator?

<expr> → <operator> <operands>

<operator> → PLUS | MINUS | TIMES | DIVIDE

<operands> → <number> | <number> <operands>

<number> → INT | FLOAT

EBNF for Prefix Calculator?

Is the production below already in EBNF?

<expr> → <operator> <operands>

A. Yes

B. No

BNF for Prefix Calculator?

Is the production below already in EBNF?

<operator> → PLUS | MINUS | TIMES | DIVIDE

A. Yes

B. No

BNF for Prefix Calculator?

Is the production below already in EBNF?

<operands> → <number> | <number> <operands>

A. Yes

B. No

EBNF for Prefix Calculator?

EBNF?

<operands> → <number> | <number> <operands>

Right Recursion

<operands> → <number> [<operands>]

EBNF for Prefix Calculator?

Is the production below already in EBNF?

<number> → INT | FLOAT

A. Yes

B. No

EBNF for Prefix Calculator

<expr> → <operator> <operands>

<operator> → PLUS | MINUS | TIMES | DIVIDE

<operands> → <number> [<operands>]

<number> → INT | FLOAT

EBNF for MUFL 1.0

<expr> -> OPENPAREN OPERATOR <operands> CLOSEPAREN |
NUMBER

<operands> -> <expr> | <expr> <operands>

<expr> -> OPENPAREN OPERATOR <operands> CLOSEPAREN |
NUMBER

<operands> -> <expr> [<operands>]

Parsing Techniques and Tools

- A grammar written in BNF, EBNF, or syntax diagrams describes the strings of tokens that are syntactically legal
- It also describes how a parser must act to parse correctly

Parsers

- Recognizers
- Top-down parsers
- Bottom-up parsers

Recognizer

- Recognizer: a program that accepts or rejects strings based on whether they are legal strings in the language

Top-down Parser

- Expands non-terminals to match incoming tokens and directly construct a derivation
- Recursive descent parser is a top-down parser

Recursive Descent Parser

- Turn non-terminals into a group of mutually recursive functions based on the right-hand sides of the EBNFs
- Tokens are matched directly with input tokens as constructed by a scanner
- Non-terminals are interpreted as calls to the functions corresponding to the non-terminals

Recursive Descent Parser Example

1) <sentence>→<noun-phrase><verb-phrase>.
2) <noun-phrase> → <article> <noun>
3) <article> → a | the
4) <noun> → girl | dog
5) <verb-phrase> → <verb> <noun-phrase>
6) <verb> → sees | pets

```
data Token = Article String
           | Noun String
           | Verb String
           | Period
    deriving Show
```

<sentence>→<noun-phrase><verb-phrase> PERIOD

<noun-phrase> → ARTICLE NOUN

<verb-phrase> → VERB <noun-phrase>

Recursive Descent Parser Example

<sentence>→<noun-phrase><verb-phrase> PERIOD

<noun-phrase> → > ARTICLE NOUN

<verb-phrase> → VERB <noun-phrase>

```
data Token = Article String
           | Noun String
           | Verb String
           | Period
  deriving Show
```

Recursive Descent Parser Example

<verb-phrase> → VERB <noun-phrase>

```
def verb_phrase():
    if token == 'VERB':
        match() # advance to the next token
        noun_phrase()
    else:
        error …
```

Non-terminals are interpreted as calls to the functions corresponding to the non-terminals

Recognizer Example

<verb-phrase> → VERB <noun-phrase>

```
verbphrase:: [Token] -> (Bool, [Token])
verbphrase (Verb _:rest) = nounphrase rest
verbphrase _ = error "Verb phrase expected"
```

```
data Token = Article String
           | Noun String
           | Verb String
           | Period
  deriving Show
```

Recognizer Example

<noun-phrase> → > ARTICLE NOUN

```
nounphrase:: [Token] -> (Bool, [Token])
nounphrase (Article _:Noun _:rest) = (True, rest)
nounphrase _ = error "Noun phrase expected"
```

```
data Token = Article String
           | Noun String
           | Verb String
           | Period
  deriving Show
```

Recognizer Example

<sentence>→<noun-phrase><verb-phrase> PERIOD

```
sentence:: [Token] -> Bool
sentence [] = False
sentence ts = let (np, r1) = nounphrase ts
                  (vp, r2) = verbphrase r1
              in case r2 of
                  Period:[] -> np && vp  -- ok
                  _ -> error "Period expected at the end"
```

Recursive Descent Parser Example

> sentence [Article "the", Noun "girl", Verb "sees", Article "a", Noun "dog", Period]

True

> sentence [Article "the", Noun "girl", Verb "sees", Article "a", Noun "dog"]

*** Exception: Period expected at the end

> sentence [Article "the", Noun "girl", Article "a", Noun "dog", Period]

*** Exception: Verb phrase expected

Why EBNF?

- Recursive descent parsing is based on EBNF.
- Left-recursive rules present problems

  <expr> → <expr> ADD_OP <term> | <term>

  May cause an infinite recursive loop

```
def expr():
    expr()
    ….
```

Why EBNF?

- The EBNF description expresses the recursion as a loop:

  <expr> → <term> {ADD_OP <term>}

```
def expr():
    term()
    while token is 'ADD_OP':
        term()
```

MUFL 1.0

<expr> -> OPENPAREN OPERATOR <operands> CLOSEPAREN |
NUMBER

<operands> -> <expr> [<operands>]

1. scanner
2. recognizer
3. build a parse tree
4. interpret/evaluate

MUFL 1.0 Scanner

<expr> -> OPENPAREN OPERATOR <operands> CLOSEPAREN |
NUMBER

<operands> -> <expr> [<operands>]

```
data Token = Operator Char
           | OpenParen
           | CloseParen
           | Number Float
    deriving Show
```

```
>scan "(+ 4 (* 3 5 2) 1 5 2)"
[OpenParen,Operator '+',Number 4.0,OpenParen,Operator '*',Number 3.0,Number 5.0,Number 2.0,CloseParen,Number 1.0,Number 5.0,Number 2.0,CloseParen]
```

MUFL 1.0 Scanner

<expr> -> OPENPAREN OPERATOR <operands> CLOSEPAREN |
NUMBER

<operands> -> <expr> [<operands>]

```
data Token = Operator Char
           | OpenParen
           | CloseParen
           | Number Float
    deriving Show
```

Use the *read* function to convert the string representing the number to a float.

```
somenumber :: Float
somenumber = read "100.4"
```

MUFL 1.0 Recognizer

<expr> -> OPENPAREN OPERATOR <operands> CLOSEPAREN |
NUMBER

<operands> -> <expr> [<operands>]

```
recognizer :: [Token] -> Bool
expr :: [Token] -> (Bool, [Token])
operands :: [Token] -> (Bool, [Token])

check = recognizer.scan
```

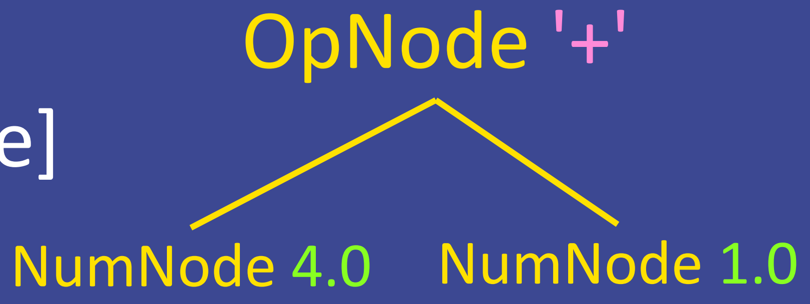
```
> check "(+ 4 (* 3 5 2) 1 5 2)"
True
> check "(+) 4 (* 3 5 2) 1 5 2)"
*** Exception: unexpected )
```

MUFL 1.0 Parse Trees

```
<expr> -> OPENPAREN OPERATOR <operands> CLOSEPAREN |
          NUMBER

<operands> ->  <expr> [<operands>]
```

```
data ParseTree = NumNode Float
               | OpNode Char [ParseTree]
     deriving Show
```




```
> parse  "(+ 4 1)"
OpNode '+' [NumNode 4.0,NumNode 1.0]
```

MUFL 1.0 Parse Trees

<expr> -> OPENPAREN OPERATOR <operands> CLOSEPAREN |
NUMBER

<operands> -> <expr> [<operands>]

build :: [Token] -> ParseTree

pexpr :: [Token] -> (ParseTree, [Token])

poperands :: [Token] -> ([ParseTree], [Token])

```
> parse  "(+ 4 (* 3 5 2) 1 5 2)"
OpNode '+' [NumNode 4.0,OpNode '*' [NumNode 3.0,NumNode 5.0,NumNode 2.0],NumNode 1.0,NumNode 5.0,NumNode 2.0]
```

parse = build.scan

MUFL 1.0 Interpreter

<expr> -> OPENPAREN OPERATOR <operands> CLOSEPAREN |
NUMBER

<operands> -> <expr> [<operands>]

eval :: ParseTree -> Float

interpret = eval.build.scan

HINT: fold and map are useful here

```
> interpret "(+ 4 (* 3 5 2) 1 5 2)"
42.0
> interpret "(- 4)"
-4.0
> interpret "(/ 2)"
0.5
> interpret "(* 2)"
2.0
> interpret "(+ 3)"
3.0
```