# Lecture 5 – Parsing

Violet Ka I Pun

violet@ifi.uio.no

# What is parsing?

**Parsing**

- A process of analysing a piece of text to determine its *syntactic structure*.

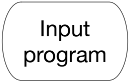
# What is parsing?

**Parsing**

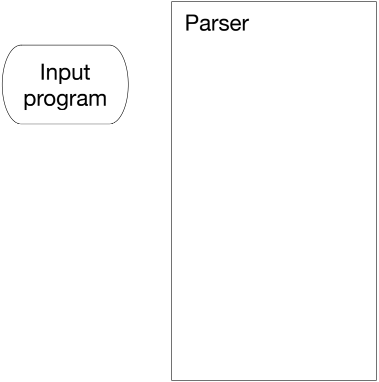
- A process of analysing a piece of text to determine its *syntactic structure*.

**Parser**

- A program that makes such syntactic analyses
  - takes some input data, usually text (e.g., program)
  - builds a data structure, like parse tree, abstract syntax tree

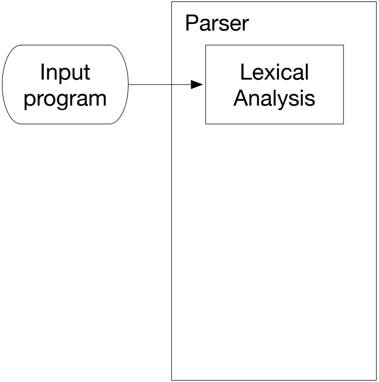

Input
program

# Parsing a program



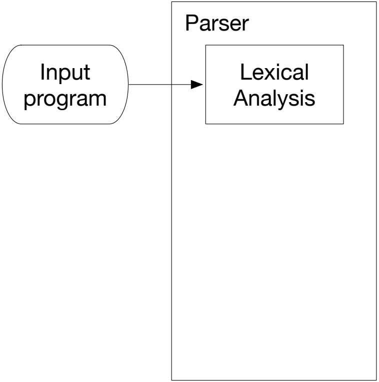

**Two steps**

# Parsing a program




**Lexical Analysis (lexer)**

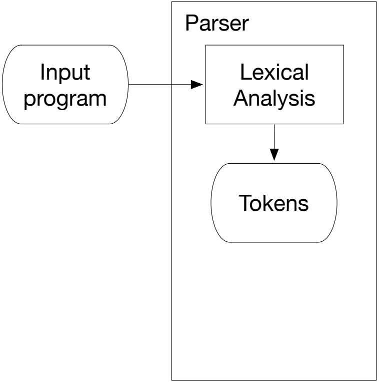
# Parsing a program




**Lexical Analysis (lexer)**

– Split into meaningful terminals

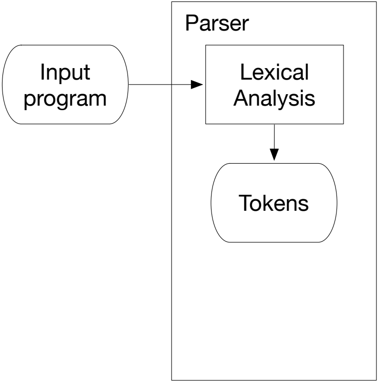
# Parsing a program




**Lexical Analysis (lexer)**

– Split into meaningful terminals

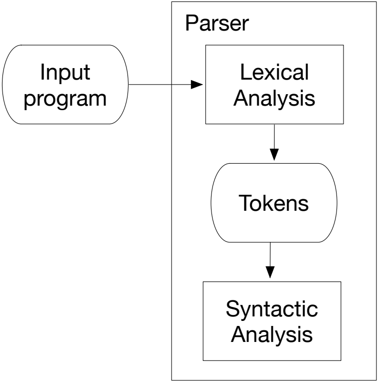
# Parsing a program




**Lexical Analysis (lexer)**

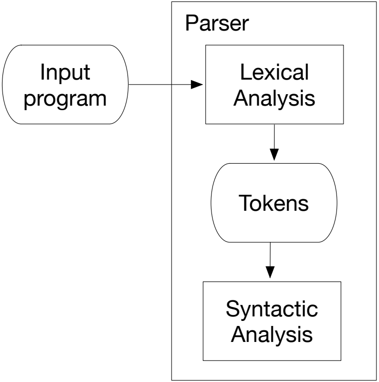
– Split into meaningful terminals

E.g.: "3 + 2" $\rightarrow$ "3", "+", "2"




**Syntactic Analysis**

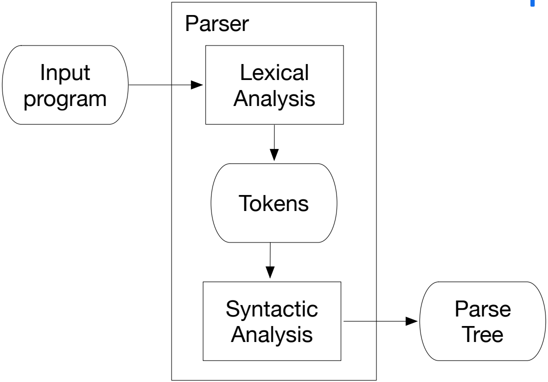
# Parsing a program




**Syntactic Analysis**

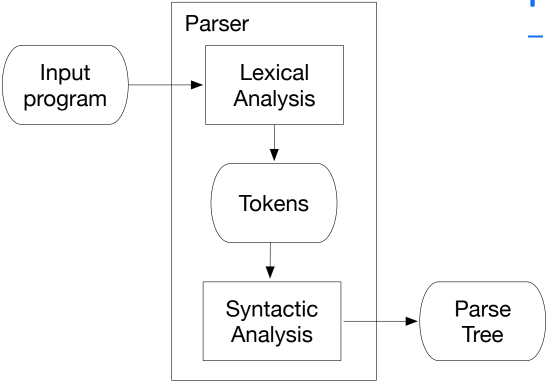
– Check whether the tokens form an illegal expression

# Parsing a program




**Parse tree**

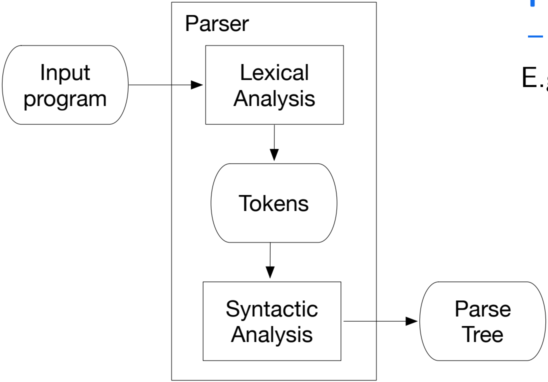
# Parsing a program




**Parse tree**

– A data structure

# Parsing a program

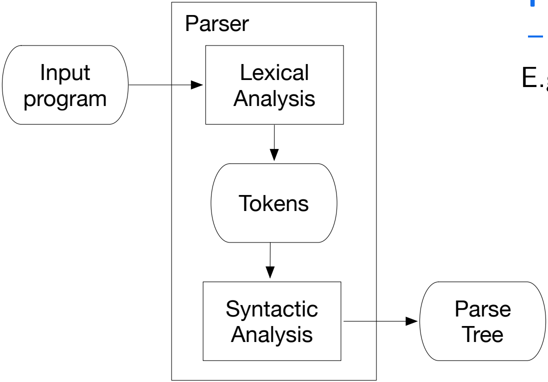



**Parse tree**

– A data structure

E.g.: "3", "+", "2"

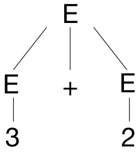
# Parsing a program




**Parse tree**

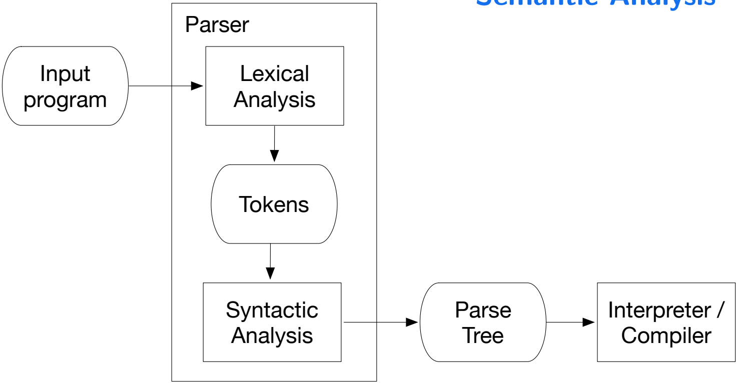
– A data structure

E.g.: “3”, “+”, “2”

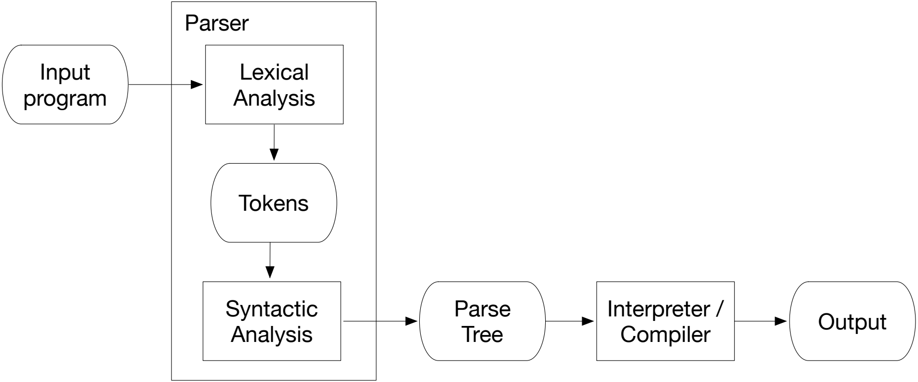


# Parsing a program

**Semantic Analysis**



# Parsing a program



# Parsers

**Task**

- To decide whether an input *can be* derived and *how* to derive from a starting symbol

# Parsers

**Task**

- To decide whether an input *can be* derived and *how* to derive from a starting symbol

Mainly two ways:

- Top-down
- Bottom-up

**Task**

- To decide whether an input *can be* derived and *how* to derive from a starting symbol

Mainly two ways:

- Top-down
  - start from the starting non-terminal
  - pick a production rule and try to match the input string
  - attempt to find the leftmost derivation
  - may require *backtracking*
  - predictive grammar $\leftarrow$ no backtracking
- Bottom-up

**Task**

- To decide whether an input *can be* derived and *how* to derive from a starting symbol

Mainly two ways:

- Top-down
  - start from the starting non-terminal
  - pick a production rule and try to match the input string
  - attempt to find the leftmost derivation
  - may require *backtracking*
  - predictive grammar $\leftarrow$ no backtracking
- Bottom-up
  - start from the input string
  - locate the most basic elements, then
  - find the elements containing those identified ones, until
  - the starting non-terminal is reached

To parse "a * b + c"

expression ::= term | expression add term

term ::= operand | term multi operand

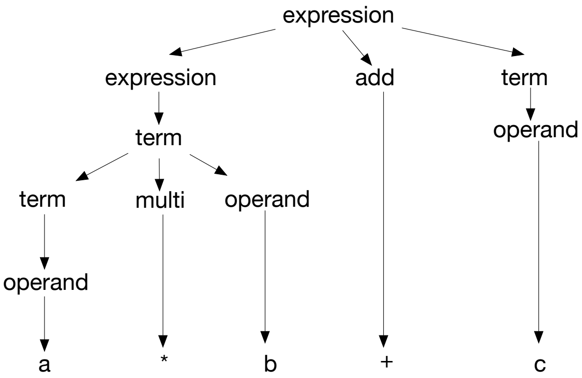
operand ::= a | b | c

add ::= +

mulit ::= *

# Top-down parsing

To parse "a * b + c"

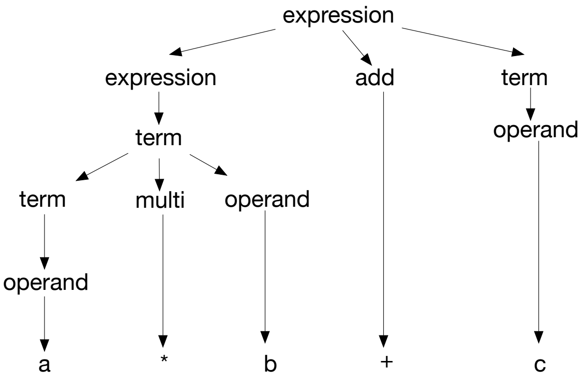



expression ::= term | expression add term
term ::= operand | term multi operand
operand ::= a | b | c

add ::= +
mulit ::= *

# Top-down parsing

To parse "a * b + c"




– Recursive descent parser
– LL parser

expression ::= term | expression add term

term ::= operand | term multi operand

operand ::= a | b | c

add ::= +

mulit ::= *

To parse "a * b + c"

expression ::= term | expression add term

term ::= operand | term multi operand

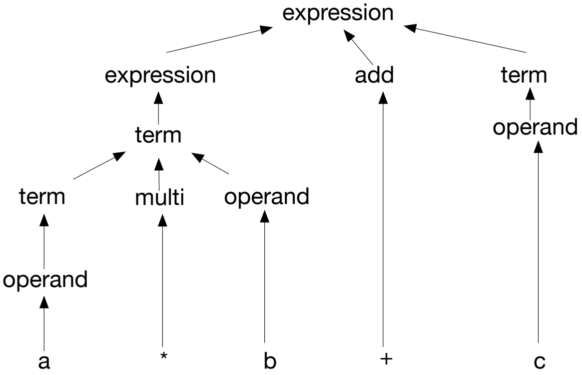
operand ::= a | b | c

add ::= +

mulit ::= *

# Bottom-up parsing

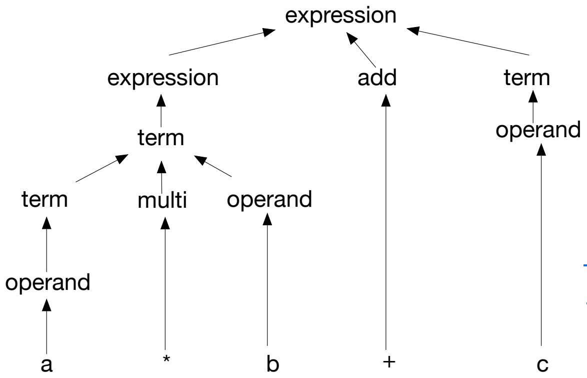
To parse "a * b + c"




expression ::= term | expression add term     add ::= +
term ::= operand | term multi operand     mulit ::= *
operand ::= a | b | c

# Bottom-up parsing

To parse "a * b + c"




– Shift-reduce parser
– LR parser

expression ::= term | expression add term

term ::= operand | term multi operand

operand ::= a | b | c

add ::= +

mulit ::= *

# Some useful functions

```
takeWhile ::
```

# Some useful functions

```
takeWhile ::
takeWhile (< 4) [1,2,1,4,1,2,3,4]
```

# Some useful functions

```
takeWhile ::
takeWhile (< 4) [1,2,1,4,1,2,3,4]......................[1,2,1]
```

# Some useful functions

```
takeWhile ::
takeWhile (< 4) [1,2,1,4,1,2,3,4].......................[1,2,1]
takeWhile (< 0) [1,2,3]
```

# Some useful functions

```
takeWhile ::
takeWhile (< 4) [1,2,1,4,1,2,3,4]......................[1,2,1]
takeWhile (< 0) [1,2,3] ..................................[]
```

# Some useful functions

```
takeWhile ::  (a -> Bool) -> [a] -> [a]
takeWhile (< 4) [1,2,1,4,1,2,3,4].....................[1,2,1]
takeWhile (< 0) [1,2,3] ................................[]
```

# Some useful functions

```
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile (< 4) [1,2,1,4,1,2,3,4].....................[1,2,1]
takeWhile (< 0) [1,2,3] ................................[]

takeWhile _ []          =  []
```

# Some useful functions

```
takeWhile ::  (a -> Bool) -> [a] -> [a]
takeWhile (< 4) [1,2,1,4,1,2,3,4]......................[1,2,1]
takeWhile (< 0) [1,2,3] .................................[]

takeWhile _ []          =  []
takeWhile p (x:xs)
            | p x       =  x : takeWhile p xs
            | otherwise =  []
```

# Some useful functions

```
takeWhile ::  (a -> Bool) -> [a] -> [a]
takeWhile (< 4) [1,2,1,4,1,2,3,4]......................[1,2,1]
takeWhile (< 0) [1,2,3] .................................[]

takeWhile _ []           =  []
takeWhile p (x:xs)
            | p x        =  x : takeWhile p xs
            | otherwise =  []

dropWhile ::
```

## Some useful functions

```
takeWhile ::  (a -> Bool) -> [a] -> [a]
takeWhile (< 4) [1,2,1,4,1,2,3,4].....................[1,2,1]
takeWhile (< 0) [1,2,3] .................................[]

takeWhile _ []           =  []
takeWhile p (x:xs)
            | p x        =  x : takeWhile p xs
            | otherwise  =  []

dropWhile ::
dropWhile (< 3) [1,2,3,4,5,1,2,3]
```

# Some useful functions

```
takeWhile ::  (a -> Bool) -> [a] -> [a]
takeWhile (< 4) [1,2,1,4,1,2,3,4]....................[1,2,1]
takeWhile (< 0) [1,2,3] ..................................[]

takeWhile _ []          =  []
takeWhile p (x:xs)
            | p x       =  x : takeWhile p xs
            | otherwise =  []

dropWhile ::
dropWhile (< 3) [1,2,3,4,5,1,2,3]..............[3,4,5,1,2,3]
```

# Some useful functions

```
takeWhile ::  (a -> Bool) -> [a] -> [a]
takeWhile (< 4) [1,2,1,4,1,2,3,4]....................[1,2,1]
takeWhile (< 0) [1,2,3] ..................................[]

takeWhile _ []          =  []
takeWhile p (x:xs)
            | p x       =  x : takeWhile p xs
            | otherwise =  []

dropWhile ::
dropWhile (< 3) [1,2,3,4,5,1,2,3]..............[3,4,5,1,2,3]
dropWhile (< 0) [1,2,3]
```

# Some useful functions

```
takeWhile ::  (a -> Bool) -> [a] -> [a]
takeWhile (< 4) [1,2,1,4,1,2,3,4]....................[1,2,1]
takeWhile (< 0) [1,2,3] ...............................[]

takeWhile _ []          =  []
takeWhile p (x:xs)
            | p x       =  x : takeWhile p xs
            | otherwise =  []

dropWhile ::
dropWhile (< 3) [1,2,3,4,5,1,2,3]..............[3,4,5,1,2,3]
dropWhile (< 0) [1,2,3]...........................[1,2,3]
```

# Some useful functions

```
takeWhile ::  (a -> Bool) -> [a] -> [a]
takeWhile (< 4) [1,2,1,4,1,2,3,4]...................[1,2,1]
takeWhile (< 0) [1,2,3] .............................[]

takeWhile _ []          =  []
takeWhile p (x:xs)
             | p x       =  x : takeWhile p xs
             | otherwise =  []

dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile (< 3) [1,2,3,4,5,1,2,3].............[3,4,5,1,2,3]
dropWhile (< 0) [1,2,3]............................[1,2,3]
```

# Some useful functions

```
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile (< 4) [1,2,1,4,1,2,3,4].......................[1,2,1]
takeWhile (< 0) [1,2,3] .................................[]

takeWhile _ []          =  []
takeWhile p (x:xs)
          | p x         =  x : takeWhile p xs
          | otherwise =  []

dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile (< 3) [1,2,3,4,5,1,2,3]...............[3,4,5,1,2,3]
dropWhile (< 0) [1,2,3]............................[1,2,3]

dropWhile _ []          =  []
```

# Some useful functions

```
takeWhile ::  (a -> Bool) -> [a] -> [a]
takeWhile (< 4) [1,2,1,4,1,2,3,4]...................[1,2,1]
takeWhile (< 0) [1,2,3] ...............................[]

takeWhile _ []          =  []
takeWhile p (x:xs)
          | p x         =  x : takeWhile p xs
          | otherwise =  []

dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile (< 3) [1,2,3,4,5,1,2,3]..............[3,4,5,1,2,3]
dropWhile (< 0) [1,2,3]............................[1,2,3]

dropWhile _ []          =  []
dropWhile p xs@(x:xs')
          | p x         =  dropWhile p xs'
          | otherwise =  xs
```

# Some useful functions

```
id ::
```

# Some useful functions

```
id ::
id 3
```

# Some useful functions

```
id ::
id 3.......................................................3
```

# Some useful functions

```
id ::
id 3.......................................................3
id (1,2)
```

# Some useful functions

```
id ::
id 3......................................................3
id (1,2) ..........................................(1,2)
```

# Some useful functions

```
id :: a -> a                                        identity function
id 3.....................................................3
id (1,2)............................................(1,2)
```

# Some useful functions

```
id :: a -> a                                        identity function
id 3...................................................3
id (1,2)...........................................(1,2)

const ::
```

# Some useful functions

```
id :: a -> a                                        identity function
id 3.....................................................3
id (1,2)..............................................(1,2)

const ::
const 1 2
```

# Some useful functions

```
id :: a -> a                          identity function
id 3..........................................3
id (1,2)..................................(1,2)

const ::
const 1 2.....................................1
```

# Some useful functions

```
id :: a -> a                                   identity function
id 3..............................................3
id (1,2).....................................(1,2)

const ::
const 1 2.........................................1
const 10 'a'
```

# Some useful functions

```
id :: a -> a                                   identity function
id 3..............................................3
id (1,2)......................................(1,2)

const ::
const 1 2.........................................1
const 10 'a'.....................................10
```

# Some useful functions

```
id :: a -> a                                  identity function
id 3........................................................3
id (1,2)............................................(1,2)

const :: a -> b -> a                          constant function
const 1 2...................................................1
const 10 'a'...............................................10
```

# Some useful functions

```
id :: a -> a                          identity function
id 3...........................................3
id (1,2)...................................(1,2)

const :: a -> b -> a                  constant function
const 1 2......................................1
const 10 'a'..................................10

const 1 ::
```

# Some useful functions

```
id :: a -> a                                    identity function
id 3...........................................3
id (1,2)...................................(1,2)

const :: a -> b -> a                            constant function
const 1 2......................................1
const 10 'a'..................................10

const 1 :: ???
```

# Some useful functions

```
id :: a -> a                                  identity function
id 3.................................................3
id (1,2)........................................(1,2)

const :: a -> b -> a                          constant function
const 1 2............................................1
const 10 'a'........................................10

const 1 ::          b -> a
```

# Some useful functions

```
id :: a -> a                                  identity function
id 3..............................................3
id (1,2)......................................(1,2)

const :: a -> b -> a                          constant function
const 1 2.........................................1
const 10 'a'.....................................10

const 1 :: Num a => b -> a
```

# Example – a dictionary

**As a list of pairs**

`type Dict = [(Key, Value)]` for some types `Key` and `Value`

# Example – a dictionary

**As a list of pairs**

`type Dict = [(Key, Value)]` for some types `Key` and `Value`

**As a function**

`type Dict = Key -> Value`

# Example – a dictionary

**As a list of pairs**

`type Dict = [(Key, Value)]` for some types `Key` and `Value`

**As a function**

```
type Dict = Key -> Value

emptyDict ::
```

# Example – a dictionary

**As a list of pairs**

`type Dict = [(Key, Value)]` for some types `Key` and `Value`

**As a function**

```
type Dict = Key -> Value
```

```
emptyDict :: Dict
```

# Example – a dictionary

**As a list of pairs**

`type Dict = [(Key, Value)]` for some types `Key` and `Value`

**As a function**

```
type Dict = Key -> Value
```

```
emptyDict :: Dict
emptyDict =
```

# Example – a dictionary

**As a list of pairs**

`type Dict = [(Key, Value)]` for some types `Key` and `Value`

**As a function**

```
type Dict = Key -> Value
```

```
emptyDict :: Dict
emptyDict = \_ -> Nothing
```

# Example – a dictionary

**As a list of pairs**

`type Dict = [(Key, Value)]` for some types `Key` and `Value`

**As a function**

```
type Dict = Key -> Value
```

```
emptyDict :: Dict
emptyDict = const Nothing
```

# Example – a dictionary

**As a list of pairs**

`type Dict = [(Key, Value)]` for some types `Key` and `Value`

**As a function**

```
type Dict = Key -> Maybe Value
```

```
emptyDict :: Dict
emptyDict = const Nothing
```

# Example – a dictionary

### As a list of pairs

`type Dict = [(Key, Value)]` for some types `Key` and `Value`

### As a function

```
type Dict = Key -> Maybe Value
```

```
emptyDict :: Dict
emptyDict = const Nothing
```

```
lookup ::
```

# Example – a dictionary

**As a list of pairs**

`type Dict = [(Key, Value)]` for some types `Key` and `Value`

**As a function**

```
type Dict = Key -> Maybe Value
```

```
emptyDict :: Dict
emptyDict = const Nothing
```

```
lookup :: Dict -> Key ->
```

# Example – a dictionary

### As a list of pairs

`type Dict = [(Key, Value)]` for some types `Key` and `Value`

### As a function

```
type Dict = Key -> Maybe Value
```

```
emptyDict :: Dict
emptyDict = const Nothing
```

```
lookup :: Dict -> Key -> Maybe Value
```

### As a list of pairs

`type Dict = [(Key, Value)]` for some types `Key` and `Value`

### As a function

```
type Dict = Key -> Maybe Value

emptyDict :: Dict
emptyDict = const Nothing

lookup :: Dict -> Key -> Maybe Value
lookup d k =
```

# Example – a dictionary

**As a list of pairs**

`type Dict = [(Key, Value)]` for some types `Key` and `Value`

**As a function**

```
type Dict = Key -> Maybe Value
```

```
emptyDict :: Dict
emptyDict = const Nothing
```

```
lookup :: Dict -> Key -> Maybe Value
lookup d k = d k
```

# Example – a dictionary

**As a list of pairs**

`type Dict = [(Key, Value)]` for some types `Key` and `Value`

**As a function**

```
type Dict = Key -> Maybe Value

emptyDict :: Dict
emptyDict = const Nothing

lookup :: Dict -> Key -> Maybe Value
lookup d k = d k

add ::
```

# Example – a dictionary

**As a list of pairs**

`type Dict = [(Key, Value)]` for some types `Key` and `Value`

**As a function**

```
type Dict = Key -> Maybe Value

emptyDict :: Dict
emptyDict = const Nothing

lookup :: Dict -> Key -> Maybe Value
lookup d k = d k

add :: Dict -> Key -> Value ->
```

# Example – a dictionary

**As a list of pairs**

`type Dict = [(Key, Value)]` for some types `Key` and `Value`

**As a function**

```
type Dict = Key -> Maybe Value
```

```
emptyDict :: Dict
emptyDict = const Nothing
```

```
lookup :: Dict -> Key -> Maybe Value
lookup d k = d k
```

```
add :: Dict -> Key -> Value -> Dict
```

# Example – a dictionary

**As a list of pairs**

`type Dict = [(Key, Value)]` for some types `Key` and `Value`

**As a function**

```
type Dict = Key -> Maybe Value

emptyDict :: Dict
emptyDict = const Nothing

lookup :: Dict -> Key -> Maybe Value
lookup d k = d k

add :: Dict -> Key -> Value -> Dict
add d k v =
```

# Example – a dictionary

**As a list of pairs**

`type Dict = [(Key, Value)]` for some types `Key` and `Value`

**As a function**

```
type Dict = Key -> Maybe Value
```

```
emptyDict :: Dict
emptyDict = const Nothing
```

```
lookup :: Dict -> Key -> Maybe Value
lookup d k = d k
```

```
add :: Dict -> Key -> Value -> Dict
add d k v = \x ->
```

# Example – a dictionary

**As a list of pairs**

`type Dict = [(Key, Value)]` for some types `Key` and `Value`

**As a function**

```
type Dict = Key -> Maybe Value

emptyDict :: Dict
emptyDict = const Nothing

lookup :: Dict -> Key -> Maybe Value
lookup d k = d k

add :: Dict -> Key -> Value -> Dict
add d k v = \x -> if x == k
```

# Example – a dictionary

**As a list of pairs**

`type Dict = [(Key, Value)]` for some types `Key` and `Value`

**As a function**

```
type Dict = Key -> Maybe Value

emptyDict :: Dict
emptyDict = const Nothing

lookup :: Dict -> Key -> Maybe Value
lookup d k = d k

add :: Dict -> Key -> Value -> Dict
add d k v = \x -> if x == k then Just v
```

# Example – a dictionary

### As a list of pairs

`type Dict = [(Key, Value)]` for some types `Key` and `Value`

### As a function

```
type Dict = Key -> Maybe Value

emptyDict :: Dict
emptyDict = const Nothing

lookup :: Dict -> Key -> Maybe Value
lookup d k = d k

add :: Dict -> Key -> Value -> Dict
add d k v = \x -> if x == k then Just v else d x
```

# Lexical analysis – tokenize :: String -> [String]

- "012 * (3 + 11)" must be divided into meaningful substrings (tokens)

# Lexical analysis – tokenize :: String -> [String]

- "012 * (3 + 11)" must be divided into meaningful substrings (tokens)
  "012", "*", "(", "3", "+", "11", ")"

## Lexical analysis – tokenize :: String -> [String]

- "012 * (3 + 11)" must be divided into meaningful substrings (tokens)

e.g. [ "012", "*", "(", "3", "+", "11", ")" ]

# Lexical analysis – tokenize :: String -> [String]

- "012 * (3 + 11)" must be divided into meaningful substrings (tokens)

e.g. [ "012", "*", "(", "3", "+", "11", ")" ]

- tokenize [] = []

# Lexical analysis – tokenize :: String -> [String]

- "012 * (3 + 11)" must be divided into meaningful substrings (tokens)

e.g. [ "012", "*", "(", "3", "+", "11", ")" ]

- tokenize [] = []

  tokenize (' ':xs) = tokenize xs

# Lexical analysis – tokenize :: String -> [String]

- "012 * (3 + 11)" must be divided into meaningful substrings (tokens)

e.g. ["012", "*", "(", "3", "+", "11", ")" ]

- tokenize [] = []

  tokenize (' ':xs) = tokenize xs

  tokenize ('(':xs) = "(": tokenize xs

# Lexical analysis – tokenize :: String -> [String]

- "012 * (3 + 11)" must be divided into meaningful substrings (tokens)

e.g. ["012", "*", "(", "3", "+", "11", ")" ]

- tokenize [] = []

  tokenize (' ':xs) = tokenize xs

  tokenize ('(':xs) = "(": tokenize xs

  tokenize (')':xs) = ")": tokenize xs

- “012 * (3 + 11)” must be divided into meaningful substrings (tokens)

e.g. [“012”, “*”, “(”, “3”, “+”, “11”, “)” ]

- tokenize [] = []
  tokenize (' ':xs) = tokenize xs
  tokenize ('(':xs) = "(": tokenize xs
  tokenize (')':xs) = ")": tokenize xs
  tokenize (x:xs) = if isDigit x then –

- "012 * (3 + 11)" must be divided into meaningful substrings (tokens)

e.g. ["012", "*", "(", "3", "+", "11", ")" ]

- tokenize [] = []

  tokenize (' ':xs) = tokenize xs

  tokenize ('(':xs) = "(": tokenize xs

  tokenize (')':xs) = ")": tokenize xs

  tokenize (x:xs) = if isDigit x then –

  collect all digits until something which is not a digit . . .

# Lexical analysis – tokenize :: String -> [String]

- "012 * (3 + 11)" must be divided into meaningful substrings (tokens)

e.g. ["012", "*", "(", "3", "+", "11", ")" ]

- tokenize [] = []

  tokenize (' ':xs) = tokenize xs

  tokenize ('(':xs) = "(": tokenize xs

  tokenize (')':xs) = ")": tokenize xs

  tokenize (x:xs) = if isDigit x then –

  (takeWhile isDigit (x:xs)) : tokenize (dropWhile isDigit xs)

# Lexical analysis – tokenize :: String -> [String]

- "012 * (3 + 11)" must be divided into meaningful substrings (tokens)

e.g. ["012", "*", "(", "3", "+", "11", ")" ]

- tokenize [] = []

  tokenize (' ':xs) = tokenize xs

  tokenize ('(':xs) = "(": tokenize xs

  tokenize (')':xs) = ")": tokenize xs

  tokenize (x:xs) = if isDigit x then –
      (takeWhile isDigit (x:xs)) : tokenize (dropWhile isDigit xs)

  tokenize ('*':xs) = "*": tokenize xs

# Lexical analysis – tokenize :: String -> [String]

- "012 * (3 + 11)" must be divided into meaningful substrings (tokens)

e.g. ["012", "*", "(", "3", "+", "11", ")" ]

- tokenize [] = []

  tokenize (' ':xs) = tokenize xs

  tokenize ('(':xs) = "(": tokenize xs

  tokenize (')':xs) = ")": tokenize xs

  tokenize (x:xs) = if isDigit x then –
  (takeWhile isDigit (x:xs)) : tokenize (dropWhile isDigit xs)

  tokenize ('*':xs) = "*": tokenize xs

etc. ...

# Lexical analysis – tokenize :: String -> [String]

... or

... or

make a list with simple signs that are tokens:

t = "*+()" == ['*', '+', '(', ')']

... or

make a list with simple signs that are tokens:

t = “*+()” == ['*', '+', '(', ')']

and one with simple signs that separate tokens:

s = “ ” == [' '] (whitespace)

# Lexical analysis – tokenize :: String -> [String]

... or

make a list with simple signs that are tokens:

t = “*+()” == ['*', '+', '(', ')']

and one with simple signs that separate tokens:

s = “ ” == [' '] (whitespace)

tokenize [] t s = []

... or

make a list with simple signs that are tokens:

t = “*+()” == ['*', '+', '(', ')']

and one with simple signs that separate tokens:

s = “ ” == [' '] (whitespace)

tokenize [] t s = []

tokenize (x:xs) t s

# Lexical analysis – tokenize :: String -> [String]

... or

make a list with simple signs that are tokens:

t = "*+()" == ['*', '+', '(', ')']

and one with simple signs that separate tokens:

s = " " == [' '] (whitespace)

```
tokenize [] t s = []
tokenize (x:xs) t s
      | elem x t = [x] : tokenize xs t s
```

... or

make a list with <u>simple signs</u> that are tokens:

t = "*+()" == ['*', '+', '(', ')']

and one with <u>simple signs</u> that separate tokens:

s = " " == [' '] (whitespace)

```
tokenize [] t s = []
tokenize (x:xs) t s
    | elem x t = [x] : tokenize xs t s
    | elem x s = tokenize xs t s
```

... or

make a list with simple signs that are tokens:

t = "*+()" == ['*', '+', '(', ')']

and one with simple signs that separate tokens:

s = " " == [' '] (whitespace)

```
tokenize [] t s = []
tokenize (x:xs) t s
        | elem x t = [x] : tokenize xs t s
        | elem x s = tokenize xs t s
        | otherwise = (takeWhile (notin t++s) (x:xs)) :
                          tokenize (dropWhile (notin t++s) (x:xs)) t s
```

# Lexical analysis – tokenize :: String -> [String]

... or

make a list with simple signs that are tokens:

t = "*+()" == ['*', '+', '(', ')']

and one with simple signs that separate tokens:

s = " " == [' '] (whitespace)

```
tokenize [] t s = []
tokenize (x:xs) t s
      | elem x t = [x] : tokenize xs t s
      | elem x s = tokenize xs t s
      | otherwise = (takeWhile (notin t++s) (x:xs)) :
                        tokenize (dropWhile (notin t++s) (x:xs)) t s
```

there notin xs = \x -> not( elem x xs )

... or

make a list with simple signs that are tokens:

t = "*+()" == ['*', '+', '(', ')']

and one with simple signs that separate tokens:

s = " " == [' '] (whitespace)

```
tokenize [] t s = []
tokenize (x:xs) t s
        | elem x t = [x] : tokenize xs t s
        | elem x s = tokenize xs t s
        | otherwise = (takeWhile (notin t++s) (x:xs)) :
                               tokenize (dropWhile (notin t++s) (x:xs)) t s
```

there notin xs = \x -> not( elem x xs )

or two predicates `tp, sp :: Char -> Bool`

... or

make a list with simple signs that are tokens:

t = "*+()" == ['*', '+', '(', ')']

and one with simple signs that separate tokens:

s = " " == [' '] (whitespace)

```
tokenize [] t s = []
tokenize (x:xs) t s
        | elem x t = [x] : tokenize xs t s
        | elem x s = tokenize xs t s
        | otherwise = (takeWhile (notin t++s) (x:xs)) :
                              tokenize (dropWhile (notin t++s) (x:xs)) t s
```

there notin xs = \x -> not( elem x xs )

or two predicates `tp, sp :: Char -> Bool`

tokenize [] tp sp = []

... or

make a list with simple signs that are tokens:

t = "*+()" == ['*', '+', '(', ')']

and one with simple signs that separate tokens:

s = " " == [' '] (whitespace)

```
tokenize [] t s = []
tokenize (x:xs) t s
        | elem x t = [x] : tokenize xs t s
        | elem x s = tokenize xs t s
        | otherwise = (takeWhile (notin t++s) (x:xs)) :
                         tokenize (dropWhile (notin t++s) (x:xs)) t s
```

there notin xs = \x -> not( elem x xs )

or two predicates `tp, sp :: Char -> Bool`

```
tokenize [] tp sp = []
tokenize (x:xs) tp sp | tp x = [x] : tokenize xs t s
```

... or

make a list with simple signs that are tokens:
t = "*+()" == ['*', '+', '(', ')']
and one with simple signs that separate tokens:
s = " " == [' '] (whitespace)

```
tokenize [] t s = []
tokenize (x:xs) t s
        | elem x t = [x] : tokenize xs t s
        | elem x s = tokenize xs t s
        | otherwise = (takeWhile (notin t++s) (x:xs)) :
                          tokenize (dropWhile (notin t++s) (x:xs)) t s
```

there notin xs = \x -> not( elem x xs )

or two predicates `tp, sp :: Char -> Bool`

```
tokenize [] tp sp = []
tokenize (x:xs) tp sp | tp x = [x] : tokenize xs t s
                      | sp x s = tokenize xs t s
```

... or

make a list with simple signs that are tokens:

t = “*+()” == ['*', '+', '(', ')']

and one with simple signs that separate tokens:

s = “ ” == [' '] (whitespace)

```
tokenize [] t s = []
tokenize (x:xs) t s
        | elem x t = [x] : tokenize xs t s
        | elem x s = tokenize xs t s
        | otherwise = (takeWhile (notin t++s) (x:xs)) :
                          tokenize (dropWhile (notin t++s) (x:xs)) t s
```

there notin xs = \x -> not( elem x xs )

or two predicates `tp, sp :: Char -> Bool`

```
tokenize [] tp sp = []
tokenize (x:xs) tp sp | tp x = [x] : tokenize xs t s
                      | sp x s = tokenize xs t s
                      | otherwise = (takeWhile ...
```

# Recursive descent parsing

Grammar: starting symbol E,

Grammar: starting symbol E,

| | | |
|---|---|---|
| 1) | E ::= | E + T |
| 2) | | E – T |
| 3) | | T |
| 4) | T ::= | T * F |
| 5) | | T / F |
| 6) | | G |
| 8) | G::= | -G |
| 9) | | F |
| 10) | F ::= | Int |
| 11) | | (E) |

# Recursive descent parsing

Grammar: starting symbol E,

| | | | | | |
|---|---|---|---|---|---|
| 1) | E ::= | E + T | 1) | E ::= | T + E |
| 2) | | E – T | 2) | | T – E |
| 3) | | T | 3) | | T |
| 4) | T ::= | T * F | 4) | T ::= | F * T |
| 5) | | T / F | 5) | | F / T |
| 6) | | G | 6) | | G |
| 8) | G::= | -G | 8) | G::= | -G |
| 9) | | F | 9) | | F |
| 10) | F ::= | Int | 10) | F ::= | Int |
| 11) | | (E) | 11) | | (E) |

# parse :: String -> Ast

# parse :: String -> Ast

tokenize :: String -> [ String ]
then

## parse :: [String] -> Ast

tokenize :: String -> [ String ]
then

# parse :: [String] -> Ast

**Recursive descent parsing**

## Recursive descent parsing

Consider this grammar:

| | | |
|---|---|---|
| 1) | E ::= | E + T |
| 2) | | E – T |
| 3) | | T |
| 4) | T ::= | T * F |
| 5) | | T / F |
| 6) | | G |
| 8) | G::= | -G |
| 9) | | F |
| 10) | F ::= | Int |
| 11) | | (E) |

## Recursive descent parsing

Consider this grammar:

| | | | |
|---|---|---|---|
| 1) | E ::= E + T | 1) | E ::= T + E |
| 2) | E – T | 2) | T – E |
| 3) | T | 3) | T |
| 4) | T ::= T * F | 4) | T ::= F * T |
| 5) | T / F | 5) | F / T |
| 6) | G | 6) | G |
| 8) | G::= -G | 8) | G::= -G |
| 9) | F | 9) | F |
| 10) | F ::= Int | 10) | F ::= Int |
| 11) | (E) | 11) | (E) |

# parse :: [String] -> Ast

## parse :: [String] -> (Ast,[String])

# parse :: [String] -> (Ast,[String])

E ::= T + E | T

T ::= F * T | G

G ::= - G | F

F ::= int | (E)

E ::= T + E | T

T ::= F * T | G

G ::= - G | F

F ::= int | (E)

AST:

# parse :: [String] -> (Ast,[String])

E ::= T + E | T

T ::= F * T | G

G ::= - G | F

F ::= int | (E)

AST:
data Ast = F Int | G Ast | T Ast Ast | E Ast Ast

# parse :: [String] -> (Ast,[String])

E ::= T + E | T

T ::= F * T | G

G ::= - G | F

F ::= int | (E)

AST:
data Ast = V Int | N Ast | A Ast Ast | M Ast Ast

E ::= T + E | T    parseE (s)

T ::= F * T | G    parseT (s)

G ::= - G | F    parseG (s)

F ::= int | (E)    parseF (s)

AST:
data Ast = V Int | N Ast | A Ast Ast | M Ast Ast

# parse :: [String] -> (Ast,[String])

| | | |
|---|---|---|
| E ::= T + E \| T | parseE (s) | A Ast Ast |
| T ::= F * T \| G | parseT (s) | M Ast Ast |
| G ::= - G \| F | parseG (s) | N Ast |
| F ::= int \| (E) | parseF (s) | V Ast |

AST:
data Ast = V Int | N Ast | A Ast Ast | M Ast Ast

# parse :: [String] -> (Ast,[String])

# parse :: [String] -> (Ast,[String])

parseE(s) = $\qquad$ E ::= T+E | T

# parse :: [String] -> (Ast,[String])

parseE(s) = let (a,z)=parseT(s) in

E ::= T+E | T

# parse :: [String] -> (Ast,[String])

```
parseE(s) = let (a,z)=parseT(s) in
   if null z then (a,z)
```

E ::= T+E | T

# parse :: [String] -> (Ast,[String])

```
parseE(s) = let (a,z)=parseT(s) in
    if null z then (a,z)
    else if head(z) == "+" then
        let (c,rest) = parseE(tail(z)) in
```

E ::= T+E | T

# parse :: [String] -> (Ast,[String])

E ::= T+E | T

```
parseE(s) = let (a,z)=parseT(s) in
    if null z then (a,z)
    else if head(z) == "+" then
          let (c,rest) = parseE(tail(z)) in
```

# parse :: [String] -> (Ast,[String])

```
parseE(s) = let (a,z)=parseT(s) in                    E ::= T+E | T
    if null z then (a,z)
    else if head(z) == "+" then
          let (c,rest) = parseE(tail(z)) in (A a c, rest)
        else (a,z)                                    – error "expect +"
```

# parse :: [String] -> (Ast,[String])

```
parseE(s) = let (a,z)=parseT(s) in                    E ::= T+E | T
    if null z then (a,z)
    else if head(z) == "+" then
          let (c,rest) = parseE(tail(z)) in (A a c, rest)
        else (a,z)                                     – error "expect +"
 parseT(s) = let (a,z)=parseF(s) in ................. T ::= F*T | F
```

# parse :: [String] -> (Ast,[String])

```
parseE(s) = let (a,z)=parseT(s) in                        E ::= T+E | T
   if null z then (a,z)
   else if head(z) == "+" then
        let (c,rest) = parseE(tail(z)) in (A a c, rest)
      else (a,z)                                          – error "expect +"
parseT(s) = let (a,z)=parseF(s) in .................. T ::= F*T | F
   if null z then (a,z)
```

# parse :: [String] -> (Ast,[String])

```
parseE(s) = let (a,z)=parseT(s) in                              E ::= T+E | T
   if null z then (a,z)
   else if head(z) == "+" then
         let (c,rest) = parseE(tail(z)) in (A a c, rest)
        else (a,z)                                            – error "expect +"
parseT(s) = let (a,z)=parseF(s) in ..................T ::= F*T | F
   if null z then (a,z)
   else if head(z) == "*" then
         let (c,rest)=parseT(tail(z)) in
```

# parse :: [String] -> (Ast,[String])

```
parseE(s) = let (a,z)=parseT(s) in                          E ::= T+E | T
   if null z then (a,z)
   else if head(z) == "+" then
        let (c,rest) = parseE(tail(z)) in (A a c, rest)
      else (a,z)                                       – error "expect +"
 parseT(s) = let (a,z)=parseF(s) in  ................. T ::= F*T | F
   if null z then (a,z)
   else if head(z) == "*" then
        let (c,rest)=parseT(tail(z)) in
```

# parse :: [String] -> (Ast,[String])

```
parseE(s) = let (a,z)=parseT(s) in                      E ::= T+E | T
   if null z then (a,z)
   else if head(z) == "+" then
        let (c,rest) = parseE(tail(z)) in (A a c, rest)
       else (a,z)                                  – error "expect +"
parseT(s) = let (a,z)=parseF(s) in . . . . . . . . . . . . . . . . . T ::= F*T | F
   if null z then (a,z)
   else if head(z) == "*" then
        let (c,rest)=parseT(tail(z)) in (M a c, rest)
       else (a,z)                                  – error "expect *"
```

# parse :: [String] -> (Ast,[String])

```
parseE(s) = let (a,z)=parseT(s) in                      E ::= T+E | T
   if null z then (a,z)
   else if head(z) == "+" then
        let (c,rest) = parseE(tail(z)) in (A a c, rest)
       else (a,z)                                       – error "expect +"
parseT(s) = let (a,z)=parseF(s) in ................. T ::= F*T | F
   if null z then (a,z)
   else if head(z) == "*" then
        let (c,rest)=parseT(tail(z)) in (M a c, rest)
       else (a,z)                                       – error "expect *"
parseG                                                   G::= -G|F
```

# parse :: [String] -> (Ast,[String])

```
parseE(s) = let (a,z)=parseT(s) in                          E ::= T+E | T
   if null z then (a,z)
   else if head(z) == "+" then
         let (c,rest) = parseE(tail(z)) in (A a c, rest)
        else (a,z)                                      – error "expect +"
 parseT(s) = let (a,z)=parseF(s) in  . . . . . . . . . . . . . . . . . T ::= F*T | F
   if null z then (a,z)
   else if head(z) == "*" then
          let (c,rest)=parseT(tail(z)) in (M a c, rest)
         else (a,z)                                     – error "expect *"
 parseG                                                           G::= -G|F
 parseG(s) = parseF(s)
```

# parse :: [String] -> (Ast,[String])

```
parseE(s) = let (a,z)=parseT(s) in                          E ::= T+E | T
   if null z then (a,z)
   else if head(z) == "+" then
         let (c,rest) = parseE(tail(z)) in (A a c, rest)
       else (a,z)                                         – error "expect +"
 parseT(s) = let (a,z)=parseF(s) in  ..................T ::= F*T | F
   if null z then (a,z)
   else if head(z) == "*" then
          let (c,rest)=parseT(tail(z)) in (M a c, rest)
        else (a,z)                                        – error "expect *"
 parseG ("-":s) = let (a,b)=parseG(s) in (N a,b)              G::= -G|F
 parseG(s) = parseF(s)
```

# parse :: [String] -> (Ast,[String])

```
parseE(s) = let (a,z)=parseT(s) in                         E ::= T+E | T
    if null z then (a,z)
    else if head(z) == “+” then
          let (c,rest) = parseE(tail(z)) in (A a c, rest)
        else (a,z)                                      – error “expect +”
 parseT(s) = let (a,z)=parseF(s) in ................. T ::= F*T | F
    if null z then (a,z)
    else if head(z) == “*” then
          let (c,rest)=parseT(tail(z)) in (M a c, rest)
         else (a,z)                                     – error “expect *”
 parseG (“-”:s) = let (a,b)=parseG(s) in (N a,b)             G::= -G|F
 parseG(s) = parseF(s)

 parseF(“(”:s) = let(a,“)”:b)=parseE(s) in (a,b)            F::=Int|(E)
```

# parse :: [String] -> (Ast,[String])

```
parseE(s) = let (a,z)=parseT(s) in                              E ::= T+E | T
    if null z then (a,z)
    else if head(z) == "+" then
            let (c,rest) = parseE(tail(z)) in (A a c, rest)
          else (a,z)                                      – error "expect +"
 parseT(s) = let (a,z)=parseF(s) in . . . . . . . . . . . . . . . . T ::= F*T | F
    if null z then (a,z)
    else if head(z) == "*" then
             let (c,rest)=parseT(tail(z)) in (M a c, rest)
           else (a,z)                                     – error "expect *"
 parseG ("-":s) = let (a,b)=parseG(s) in (N a,b)               G::= -G|F
 parseG(s) = parseF(s)

 parseF("(":s) = let(a,")":b)=parseE(s) in (a,b)              F::=Int|(E)
 parseF(x:s) = (V(read x),s)    . . . . . . . . . . . . . . V Int |...x is a number
```

```
parseE(s) = let (a,z)=parseT(s) in                         E ::= T+E | T
   if null z then (a,z)
   else if head(z) == "+" then
        let (c,rest) = parseE(tail(z)) in (A a c, rest)
       else (a,z)                                     – error "expect +"
parseT(s) = let (a,z)=parseF(s) in ..................T ::= F*T | F
   if null z then (a,z)
   else if head(z) == "*" then
        let (c,rest)=parseT(tail(z)) in (M a c, rest)
       else (a,z)                                     – error "expect *"
parseG ("-":s) = let (a,b)=parseG(s) in (N a,b)            G::= -G|F
parseG(s) = parseF(s)

parseF("(":s) = let(a,")":b)=parseE(s) in (a,b)            F::=Int|(E)
parseF(x:s) = (V(read x),s)    ..............V Int |...x is a number
                                       ...(read x)::Int is needed
```

```
parseE(s) = let (a,z)=parseT(s) in                          E ::= T+E | T
    if null z then (a,z)
    else if head(z) == "+" then
            let (c,rest) = parseE(tail(z)) in (A a c, rest)
         else (a,z)                                          – error "expect +"
parseT(s) = let (a,z)=parseF(s) in ................. T ::= F*T | F
    if null z then (a,z)
    else if head(z) == "*" then
            let (c,rest)=parseT(tail(z)) in (M a c, rest)
         else (a,z)                                          – error "expect *"
parseG ("-":s) = let (a,b)=parseG(s) in (N a,b)             G::= -G|F
parseG(s) = parseF(s)

parseF("(":s) = let(a,")":b)=parseE(s) in (a,b)             F::=Int|(E)
parseF(x:s) = (V(read x),s)    .............. V Int |...x is a number
                                        ...(read x)::Int is needed
```

# Recursive decent parser

With **backtracking**

# Recursive decent parser

With **backtracking**

- Try each production rule

# Recursive decent parser

With **backtracking**

- Try each production rule
- May require exponential time

# Recursive decent parser

With **backtracking**

- Try each production rule
- May require exponential time
- Simple implementation cannot handle left-recursion

# Recursive decent parser

With **backtracking**

- Try each production rule
- May require exponential time
- Simple implementation cannot handle left-recursion
  infinite recursion $\rightarrow$ non-terminating

# Recursive decent parser

With **backtracking**

- Try each production rule
- May require exponential time
- Simple implementation cannot handle left-recursion
  infinite recursion → non-terminating
  - E.g., E ::= E + T | T ,     T ::= Int

# Recursive decent parser

With **backtracking**

- Try each production rule
- May require exponential time
- Simple implementation cannot handle left-recursion
  infinite recursion $\rightarrow$ non-terminating
  - E.g., E ::= E + T | T ,  T ::= Int
  - Any CFG can be transformed to a grammar that has no left-recursion

# Recursive decent parser

With **backtracking**

- Try each production rule
- May require exponential time
- Simple implementation cannot handle left-recursion
  infinite recursion $\rightarrow$ non-terminating
  - E.g., E $::=$ E $+$ T $|$ T ,    T $::=$ Int
  - Any CFG can be transformed to a grammar that has no left-recursion
  - E.g., E $::=$ T E' ,    E' $::=$ $+$ T E'$|$ $\varepsilon$

# Recursive decent parser

With **backtracking**

- Try each production rule
- May require exponential time
- Simple implementation cannot handle left-recursion
  infinite recursion $\rightarrow$ non-terminating
  - E.g., E ::= E + T | T , T ::= Int
  - Any CFG can be transformed to a grammar that has no left-recursion
  - E.g., E ::= T E' , E' ::= + T E'| $\varepsilon$
  - Right associative (!!!)

# Recursive decent parser

With **backtracking**

- Try each production rule
- May require exponential time
- Simple implementation cannot handle left-recursion
  infinite recursion $\rightarrow$ non-terminating
  - E.g., E ::= E + T | T ,   T ::= Int
  - Any CFG can be transformed to a grammar that has no left-recursion
  - E.g., E ::= T E' ,   E' ::= + T E' | $\varepsilon$
  - Right associative (!!!)

Without backtracking

- Predictive parsing

# Recursive decent parser

With **backtracking**

- Try each production rule
- May require exponential time
- Simple implementation cannot handle left-recursion
  infinite recursion $\rightarrow$ non-terminating
  - E.g., E ::= E + T | T ,  T ::= Int
  - Any CFG can be transformed to a grammar that has no left-recursion
  - E.g., E ::= T E' ,  E' ::= + T E'| $\varepsilon$
  - Right associative (!!!)

Without backtracking

- Predictive parsing
  - Only for LL(k) grammar

# Recursive decent parser

With **backtracking**

- Try each production rule
- May require exponential time
- Simple implementation cannot handle left-recursion
  infinite recursion $\rightarrow$ non-terminating
  - E.g., E ::= E + T | T ,     T ::= Int
  - Any CFG can be transformed to a grammar that has no left-recursion
  - E.g., E ::= T E' ,     E' ::= + T E'| $\varepsilon$
  - Right associative (!!!)

Without backtracking

- Predictive parsing
  - Only for LL(k) grammar
- Linear time

# LL(k)

LL grammar – allows a parser to read input from Left and construct a Leftmost derivation

# LL(k)

LL grammar – allows a parser to read input from Left and construct a Leftmost derivation

- Recursive descent, predictive parsing

LL grammar – allows a parser to read input from Left and construct a Leftmost derivation

- Recursive descent, predictive parsing

E ::= T + E | T

```
parseE(s) = let (a,z) = parseT(s) in
  if null z then (a,z)
  else if head(z) == "+" then
        let (c,rest) = parseE(tail(z)) in (A a c, rest) else...
```

LL grammar – allows a parser to read input from Left and construct a Leftmost derivation

- Recursive descent, predictive parsing

$E ::= T + E \mid T$

```
parseE(s) = let (a,z) = parseT(s) in
  if null z then (a,z)
  else if head(z) == "+" then
       let (c,rest) = parseE(tail(z)) in (A a c, rest) else...
```

# LL(k)

LL grammar – allows a parser to read input from Left and construct a Leftmost derivation

- Recursive descent, predictive parsing

```
parseE(s) = let (a,z) = parseT(s) in                    E ::= T + E | T
   if null z then (a,z)
   else if head(z) == "+" then
          let (c,rest) = parseE(tail(z)) in (A a c, rest) else...
```

# LL(k)

LL grammar – allows a parser to read input from Left and construct a Leftmost derivation

- Recursive descent, predictive parsing

  $E ::= T + E \mid T$

  ```
  parseE(s) = let (a,z) = parseT(s) in
     if null z then (a,z)
     else if head(z) == "+" then
            let (c,rest) = parseE(tail(z)) in (A a c, rest) else...
  ```

- No left-recursion! $E ::= E + P \ldots$

# LL(k)

LL grammar – allows a parser to read input from Left and construct a Leftmost derivation

- Recursive descent, predictive parsing

parseE(s) = let (a,z) = parseT(s) in $\qquad$ E ::= T + E | T
  if null z then (a,z)
  else if head(z) == "+" then
    let (c,rest) = parseE(tail(z)) in (A a c, rest) else...

- No left-recursion! $\qquad$ E :: = E + P ...

parseE(s) = let (a,z) = parseE(s) in ...

LL grammar – allows a parser to read input from Left and construct a Leftmost derivation

- Recursive descent, predictive parsing

parseE(s) = let (a,z) = parseT(s) in $\qquad$ E ::= T + E | T
  if null z then (a,z)
  else if head(z) == "+" then
      let (c,rest) = parseE(tail(z)) in (A a c, rest) else...

- No left-recursion! $\qquad$ E :: = E + P ...

parseE(s) = let (a,z) = parseE(s) in ...

- Left factored

LL grammar – allows a parser to read input from Left and construct a Leftmost derivation

- Recursive descent, predictive parsing

  parseE(s) = let (a,z) = parseT(s) in $\qquad$ E ::= T + E | T
  if null z then (a,z)
  else if head(z) == "+" then
  let (c,rest) = parseE(tail(z)) in (A a c, rest) else...

- No left-recursion! $\qquad$ E :: = E + P ...

  parseE(s) = let (a,z) = parseE(s) in ...

- Left factored

  E.g., A ::= qB | qC ← not left factored

# LL(k)

LL grammar – allows a parser to read input from Left and construct a Leftmost derivation

- Recursive descent, predictive parsing

  parseE(s) = let (a,z) = parseT(s) in $\qquad E ::= T + E \mid T$
    if null z then (a,z)
    else if head(z) == "+" then
      let (c,rest) = parseE(tail(z)) in (A a c, rest) else...
- No left-recursion! $\qquad E ::= E + P \ldots$

  parseE(s) = let (a,z) = parseE(s) in ...
- Left factored

  E.g., $A ::= qB \mid qC \leftarrow$ not left factored

  $A ::= qD \qquad D ::= B \mid C$

# LL(k)

LL(1) 1-lookahead; enough to read next symbol/token, to decide next derivation step, e.g.:

$$E ::= T + E \mid T \mid T * E \ldots$$

# LL(k)

LL(1) 1-lookahead; enough to read next symbol/token, to decide next derivation step, e.g.:

$$E ::= T + E \mid T \mid T * E \ldots$$

# LL(k)

LL(1) 1-lookahead; enough to read next symbol/token, to decide next derivation step, e.g.:

$$E ::= T + E \mid T \mid T * E \ldots$$

$$E ::= (E + F) \mid F \qquad F ::= a$$

# LL(k)

LL(1) 1-lookahead; enough to read next symbol/token, to decide next derivation step, e.g.:

$$E ::= T + E \mid T \mid T * E \ldots$$

$$E ::= (E + F) \mid F \qquad F ::= a$$

# LL(k)

LL(1) 1-lookahead; enough to read next symbol/token, to decide next derivation step, e.g.:

$$E ::= T + E \mid T \mid T * E \ldots$$

$$E ::= (E + F) \mid F \qquad F ::= a$$

LL(k) enough to read k tokens/symbols ahead

# LL(k)

LL(1) 1-lookahead; enough to read next symbol/token, to decide next derivation step, e.g.:

$$E ::= T + E \mid T \mid T * E \ldots$$

$$E ::= (E + F) \mid F \qquad F ::= a$$

LL(k) enough to read k tokens/symbols ahead

- LL(k) parsers does not require backtracking