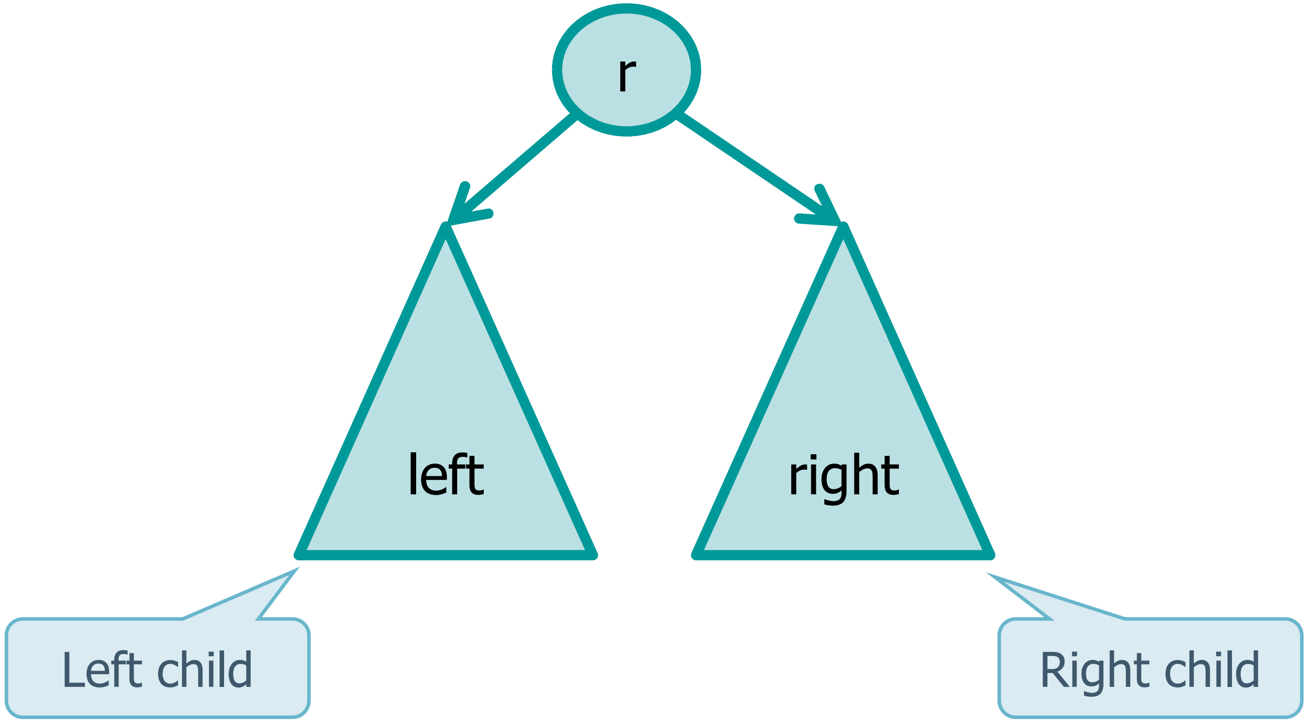
Trees

Binary Trees

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

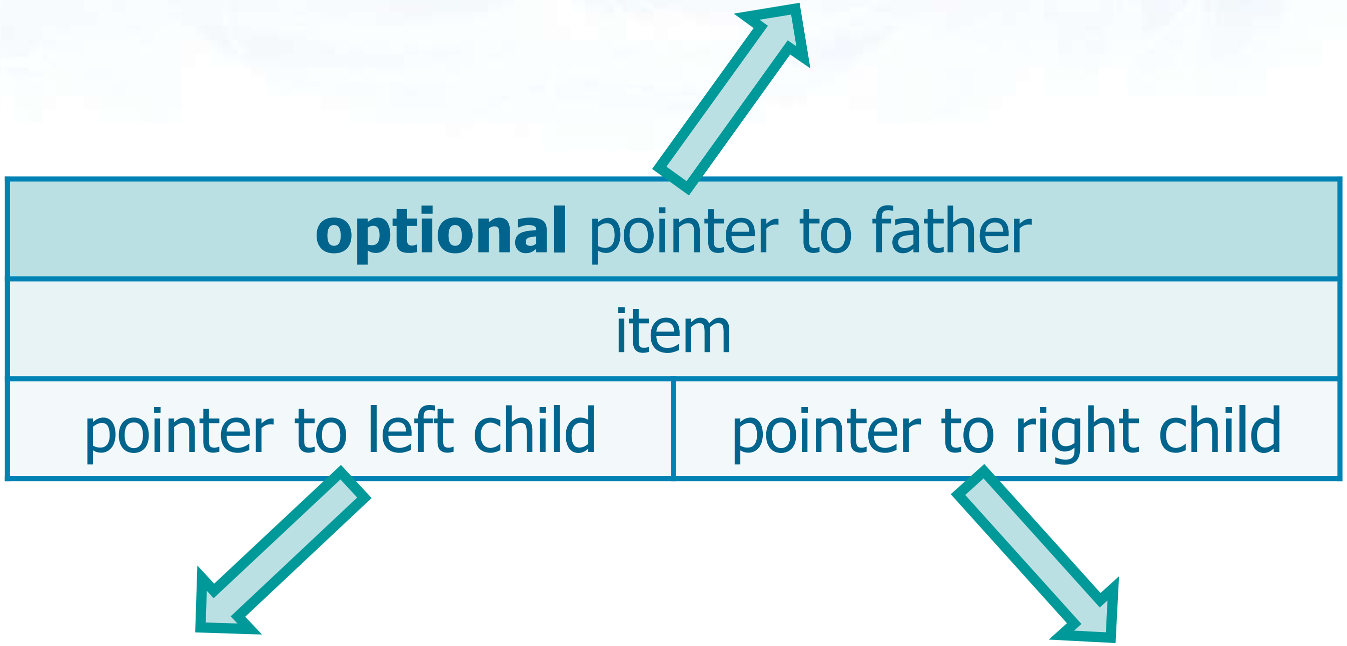
Binary Trees

- A binary tree is a tree of degree 2
- Recursive definition
  - Empty set of nodes
  - Root, left subtree, right subtree



Binary Trees

item → key
is a string
(in this section)



optional pointer to father

item

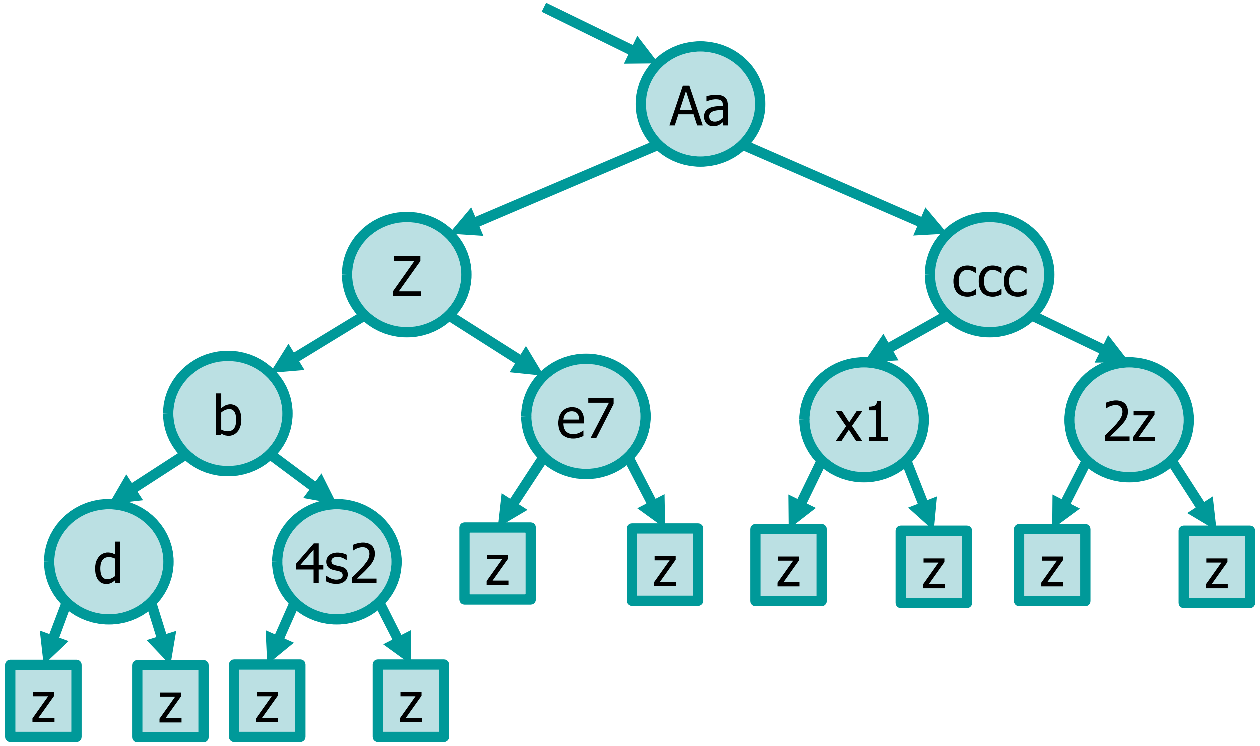
pointer to left child	pointer to right child

```
typedef struct node *link;
struct node {
  Item item;
  link l;
  link r;
};
```

ADT: We use functions
to compare keys, etc.

Binary Trees

- Tree
  - Access through pointer to root

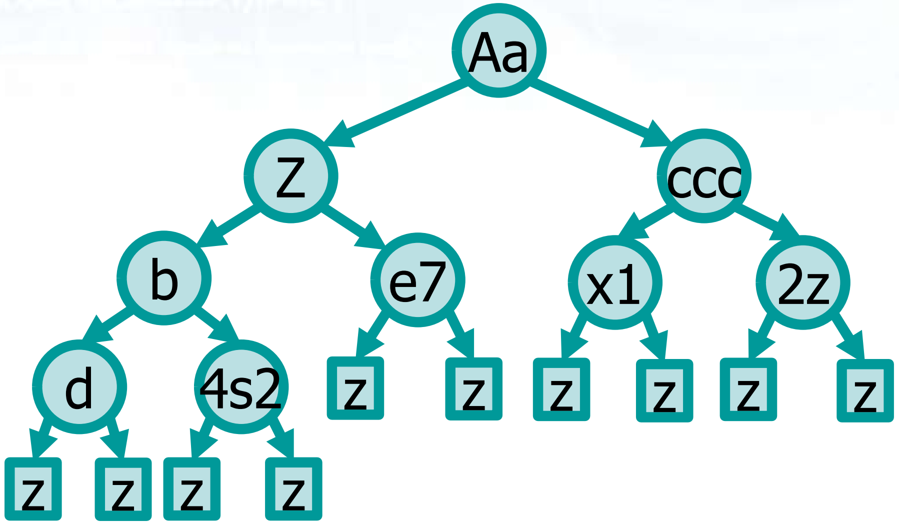


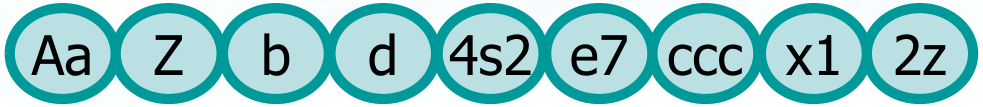
  - Dummy sentinel node z or NULL pointer

Visits

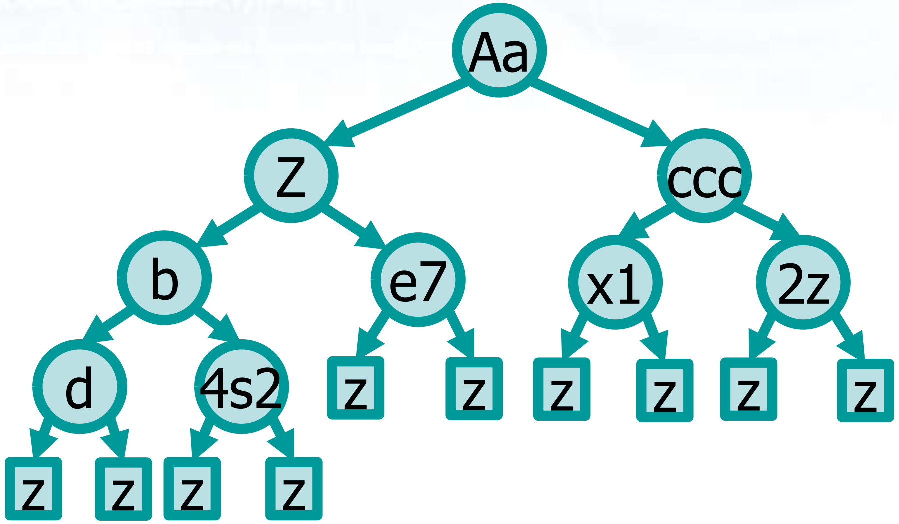
- A tree traversal or a tree visit lists the nodes according to a strategy
- Three strategies are generally used
  - Pre-order
    - **Root**, Left child (l), Right child (r)
  - In-order
    - Left child (l), **Root**, Right child (r)
  - Post-order
    - Left child (l), Right child (r), **Root**

Pre-order





Pre-order

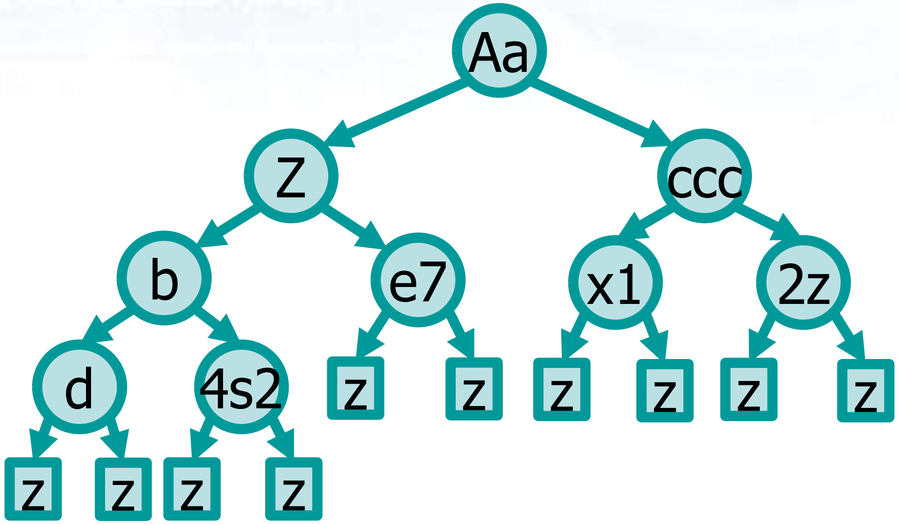


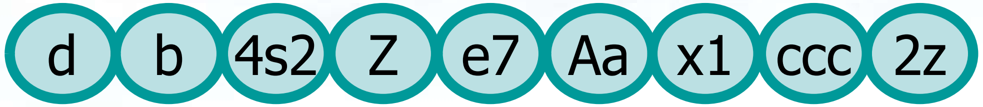
Aa Z b d 4s2 e7 ccc x1 2z

```
void preorder_r (
  link root,
  link z
) {
  if (root == z)
    return;
  item_print (root->item);
  preorder_r (root->l, z);
  preorder_r (root->r, z);
  return;
}
```

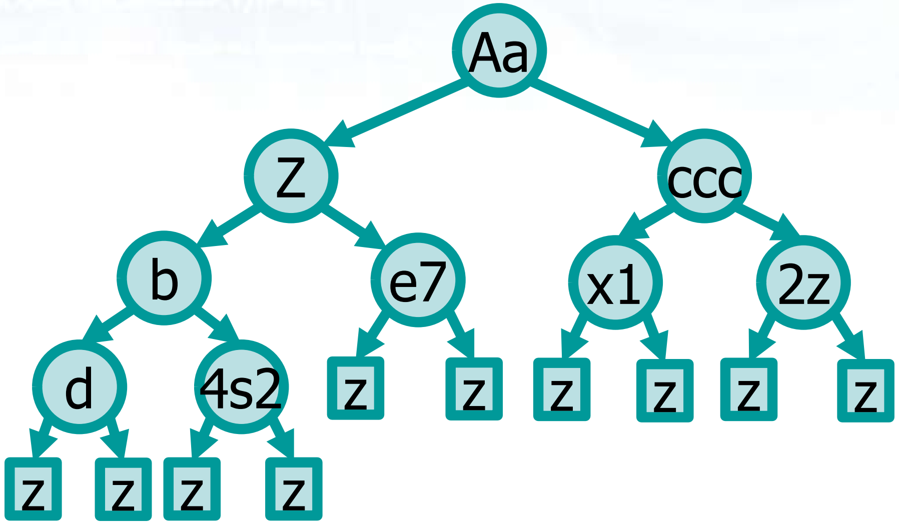
Function **item_print** writes keys

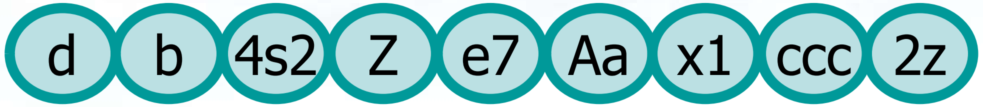
In-order





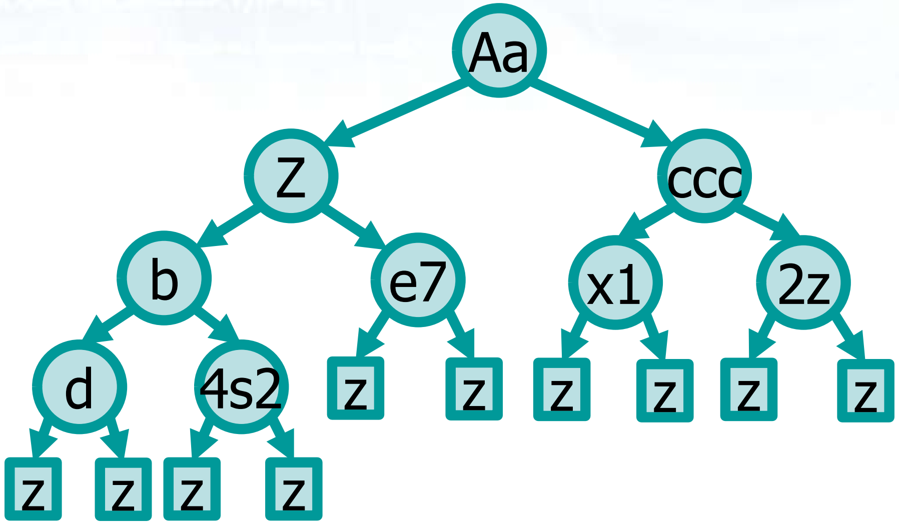
In-order

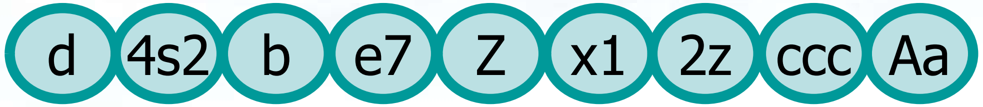



d b 4s2 Z e7 Aa x1 ccc 2z

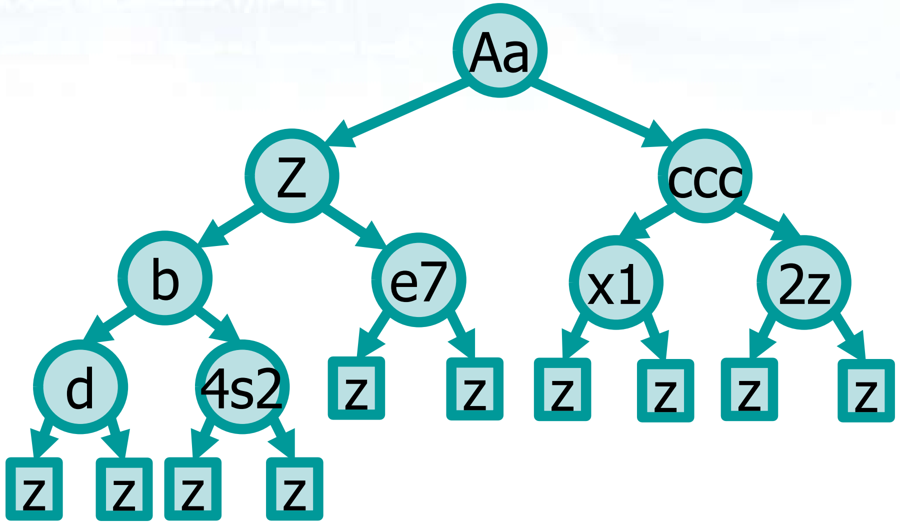
```
void inorder_r (
  link root,
  link z
){
  if (root == z)
    return;
  inorder_r (root->l, z);
  item_print (root->item);
  inorder_r (root->r, z);
  return;
}
```

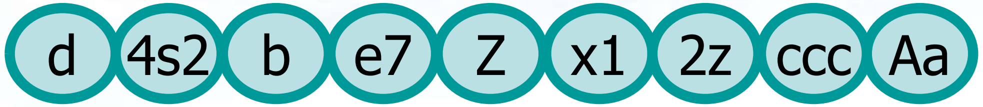
Post-order




d 4s2 b e7 Z x1 2z ccc Aa

Post-order





```
void postorder_r(
  link root,
  link z
){
  if (root == z)
    return;
  postorder_r (root->l, z);
  postorder_r (root->r, z);
  item_print (root->item);
  return;
}
```

Comparison

```
void preorder_r (
  link root,
  link z
) {
  if (root == z)
    return;
  item_print (root->item);
  preorder_r (root->l, z);
  preorder_r (root->r, z);
  return;
}
```

```
void inorder_r (
  link root,
  link z
){
  if (root == z)
    return;
  inorder_r (root->l, z);
  item_print (root->item);
  inorder_r (root->r, z);
  return;
}
```

```
void postorder_r(
  link root,
  link z
){
  if (root == z)
    return;
  postorder_r (root->l, z);
  postorder_r (root->r, z);
  item_print (root->item);
  return;
}
```

Complexity Analysis

```
void inorder_r (…){
  if (root == z)
    return;
  inorder_r (root->l, z);
  item_print (root->item);
  inorder_r (root->r, z);
  return;
}
```

Case 1

Complete tree

- $D(n) = \Theta(1)$
- $C(n) = \Theta(1)$
- $a = 2$
  - Two sub-problems
- $b = 2$
  - Overall size (of the two partitions) n-1, conservatively approximated to n, i.e., n/2 and n/2

Divide and conquer

Recurrence equation

- $T(n) = 1 + 2 \cdot T(n/2)$ $n > 1$
- $T(1) = 1$ $n = 1$

Complexity Analysis

```
void inorder_r (…){
  if (root == z)
    return;
  inorder_r (root->l, z);
  item_print (root->item);
  inorder_r (root->r, z);
  return;
}
```

- With unfolding
  - $T(n) = 1 + 2 \cdot T(n/2)$
  - $T(n/2) = 1 + 2 \cdot T(n/4)$
  - $T(n/4) = 1 + 2 \cdot T(n/8)$
  - …
  - $T(1) = 1$

Termination condition

$$\frac{n}{2^i} = 1$$

$$i = log_2 n$$

- That is
  - $T(n) = 1 + 2 \cdot (1 + 2 \cdot T(n/4))$

$$= 1 + 2 \cdot (1 + 2 \cdot (1 + 2 \cdot T(n/8)) = 1 + 2 + 4 \ldots$$

$$= \ldots$$

$$= \sum_{i=0}^{\log n} 2^i = \frac{(2^{\log n + 1} - 1)}{2 - 1} = 2 \cdot 2^{\log n - 1} = 2n\text{-}1$$

$$= \mathbf{O(n)}$$

Complexity Analysis

```
void inorder_r (…){
  if (root == z)
    return;
  inorder_r (root->l, z);
  item_print (root->item);
  inorder_r (root->r, z);
  return;
}
```

- **Case 2**
  - **Totally unbalanced tree**
  - The tree degenerates into a list
    - $D(n) = \Theta(1)$
    - $C(n) = \Theta(1)$
    - $a = 1$
    - $k_i = 1$
  - Recurrence equation
    - $T(n) = 1 + T(n-1)$ $n > 1$
    - $T(1) = 1$ $n = 1$

Complexity Analysis

```
void inorder_r (…){
  if (root == z)
    return;
  inorder_r (root->l, z);
  item_print (root->item);
  inorder_r (root->r, z);
  return;
}
```

- With unfolding
  - $T(n) = 1 + T(n-1)$
  - $T(n-1) = 1 + T(n-2)$
  - $T(n-2) = 1 + T(n-3)$
  - …
  - $T(1) = 1$
- That is
  - $T(n) = 1 + 1 + T(n-2)$

$$= 1 + 1 + 1 + T(n-3)$$

$$= \ldots$$

$$= \sum_{i=1}^{n} 1 \;= n = \mathbf{O(n)}$$

Parameter Computation

- Compute the number of nodes of a binary tree

Number of nodes

Root and Sentinel (or nothing for a termination condition checking on NULL)

```
int count (link root, link z) {
 int l, r;

 if (root == z)
   return 0;

 l = count (root->l, z);
 r = count (root->r, z);
 return (l+r+1);
}
```

Parameter Computation

- Compute the height of a binary tree

```
int height (link root, link z) {
  int u, v;

  if (root == z)
    return -1;

  u = height (root->l, z);
  v = height (root->r, z);

  if (u>v)
    return (u+1);
  else
    return (v+1);
}
```

Exercise

- Given an n-ary tree compute its
  - Number of nodes
  - Height

Node definition

```
typedef struct node *link;
struct node {
  Item item;
  int degree;
  link *child;
};
```

Array of size degree
Eventually
link child[DEGREE];

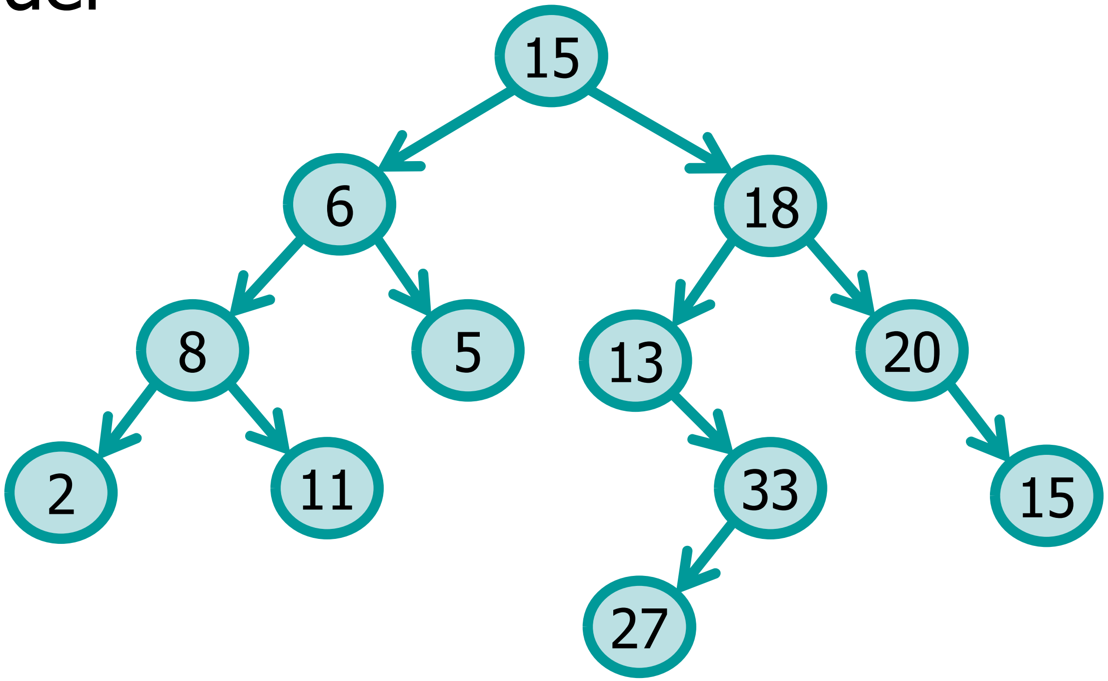
Solution

```
int count (link root, link z) {
  int i, c;
  if (root == z)
    return 0;
  for (c=0, i=0; i<root->degree; i++) {
    c = c + count (root->child[i], z);
  }
  return (c+1);
}

int height (link root, link z) {
  int i, tmp, max=-1;
  if (root == z)
    return -1;
  for (i=0; i<root->degree; i++) {
    tmp = height (root->child[i], z);
    if (tmp > max)
      max = tmp;
  }
  return (max+1);
}
```

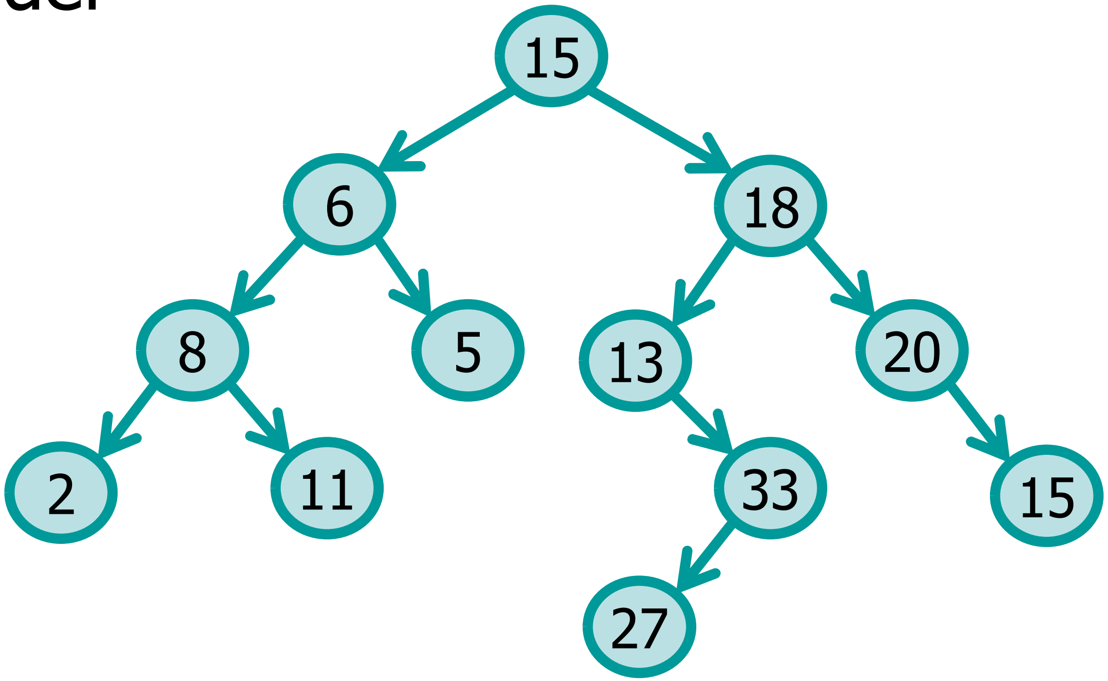
Exercise

- Given the following tree visit it in pre, in, and post-order



Solution

- Given the following tree visit it in pre, in, and post-order



- Pre-order : 15 6 8 2 11 5 18 13 33 27 20 15
- In-order : 2 8 11 6 5 15 13 27 33 18 20 15
- Post-order: 2 11 8 5 6 27 33 13 15 20 18 15

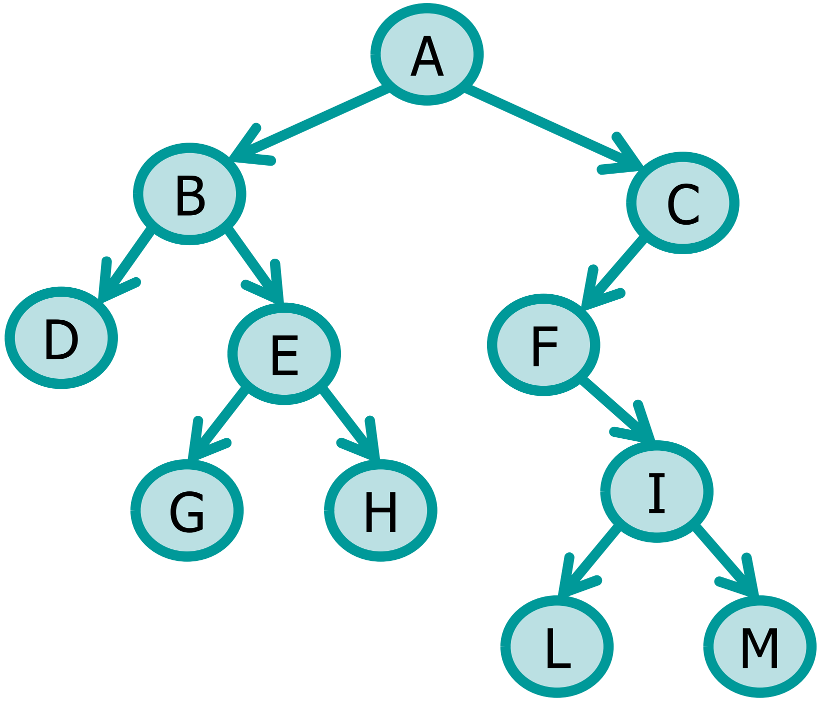
Exam: 29 January 2018

Exercise

- Consider a binary tree with 11 nodes
- Draw it considering that its pre, in and post-order visits return the following sequences
  - Pre-order: A B D E G H C F I L M
  - In-order: D B G E H A F L I M C
  - Post-order: D G H E B L M I F C A

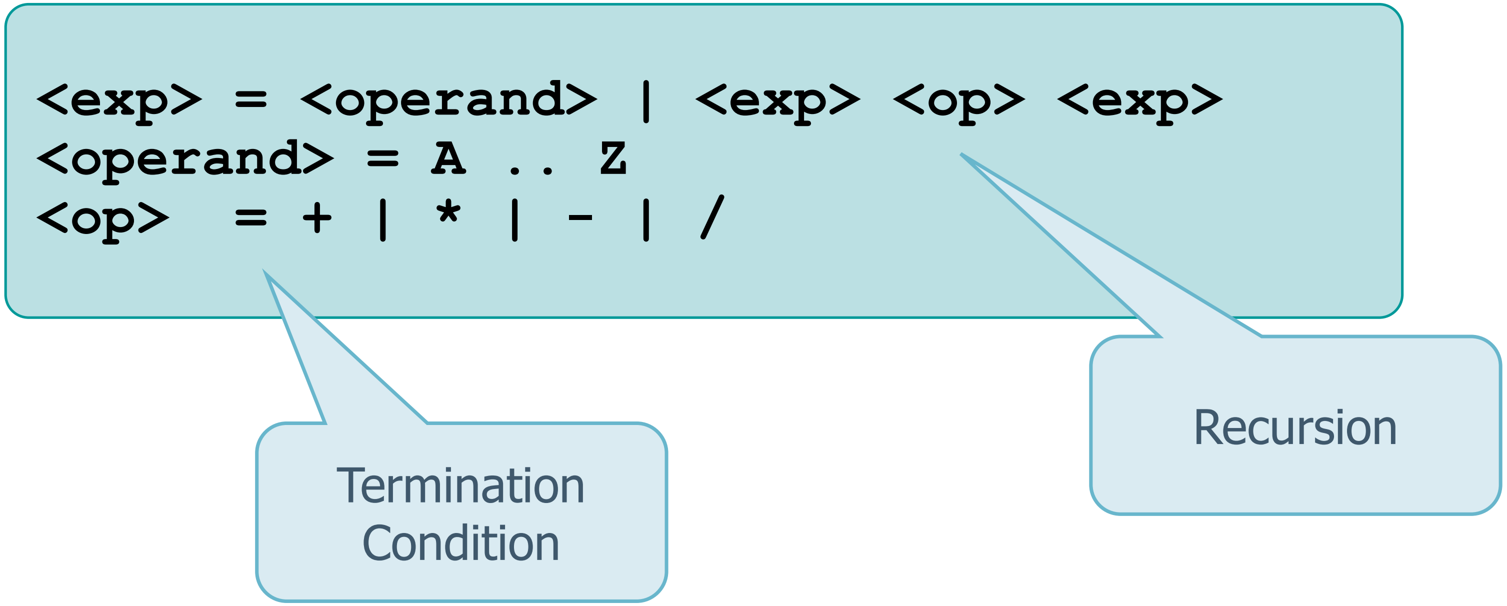
Solution

- Pre-order: A B D E G H C F I L M
- In-order: D B G E H A F L I M C
- Post-order: D G H E B L M I F C A



Application: Expressions

- Given an algebraic expression (brackets to change operator priority), it is possibile to build the corresponding tree according to the simplified grammar



```
<exp> = <operand> | <exp> <op> <exp>
<operand> = A .. Z
<op>  = + | * | - | /
```

Termination Condition

Recursion

Example

- Using the following grammar (left-hand side) parse the following equation (right-hand side)

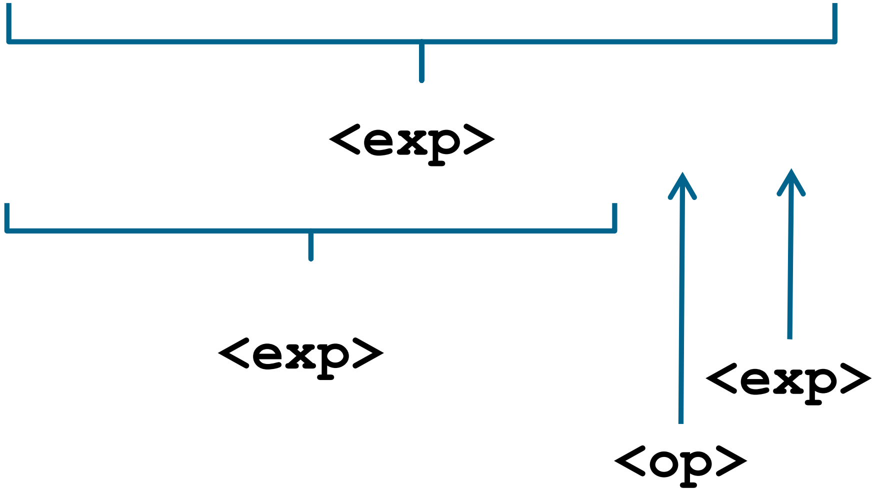
```
<exp> = <operand> | <exp> <op> <exp>
<operand> = A .. Z
<op>   = + | * | - | /
```

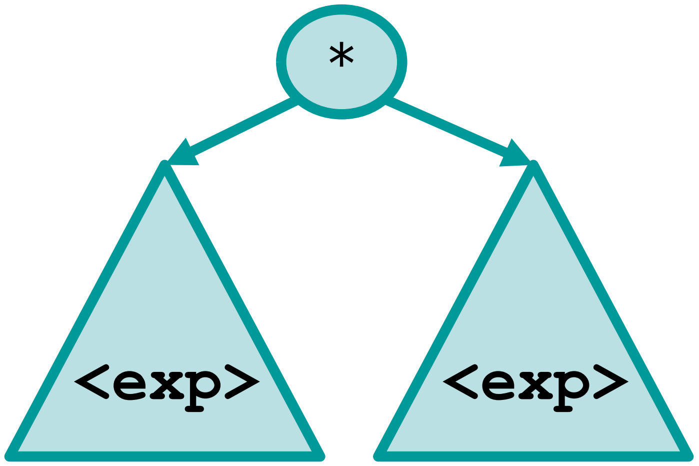
[(A + B) * (C – D)] * E

Solution: Step 1

[(A + B) * (C – D)] * E

```
<exp> = <operand> | <exp> <op> <exp>
<operand> = A .. Z
<op>  = + | * | - | /
```

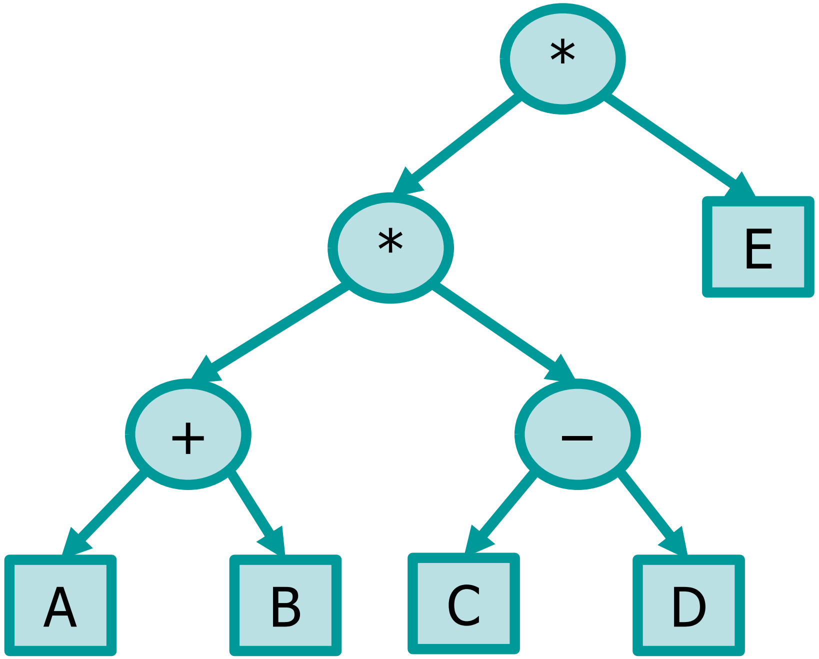




Solution

[(A + B) * (C – D)] * E

```
<exp> = <operand> | <exp> <op> <exp>
<operand> = A .. Z
<op>   = + | * | - | /
```

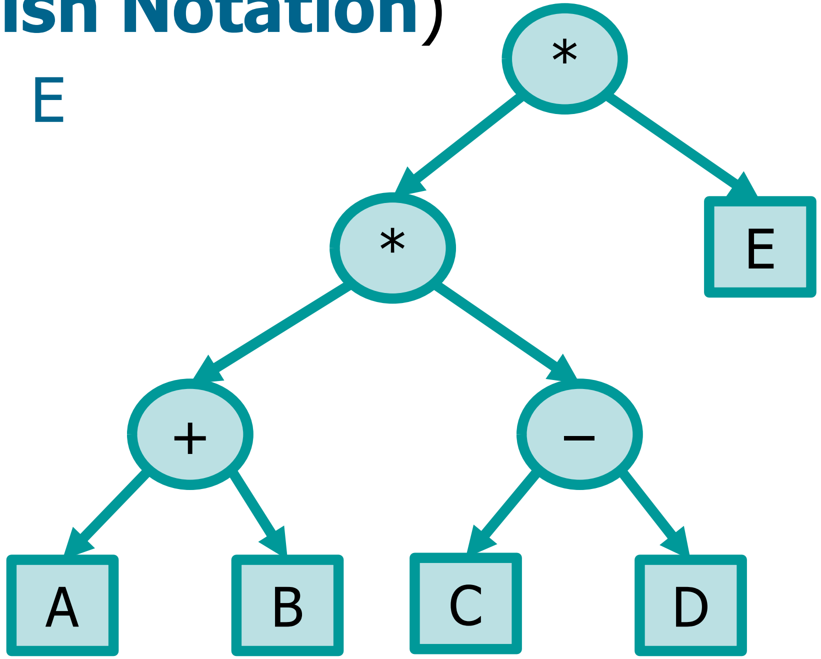


Example

[(A + B) * (C – D)] * E

- A pre-order visits returns the expression in the seldom used **prefix** form (**Polish Notation**)
  - * * + A B – C D E

Brakets are no more needed

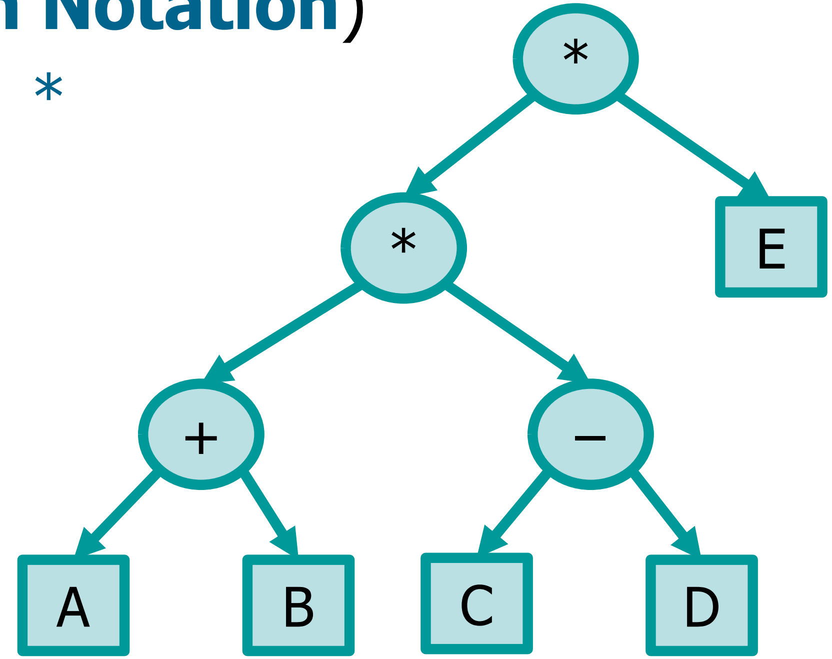


Example

[(A + B) * (C – D)] * E

- A post-order visits returns the expression in **postfix** form (**Reverse Polish Notation**)
  - A B + C D - * E *

Brakets are no more needed



Exercise

Exam: 29 January 2018

- Convert the following expressions from in-fix to post-fix and pre-fix notations

ECE students (10 credits)

(A – B) / { (C / D) + [(D / (E – F)] }

(A – B) / { (C / D) + [(D / (E – F)) * G] }

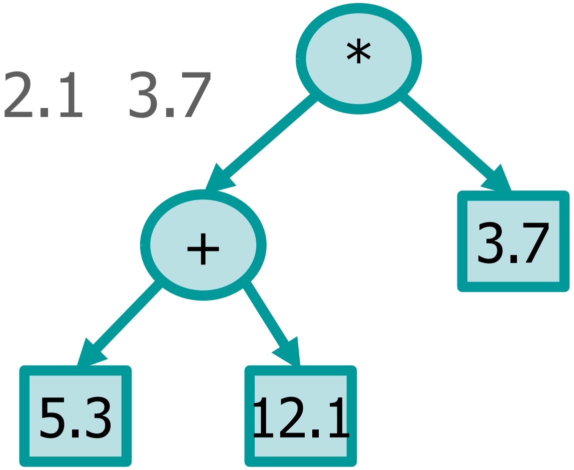
CE students (12 credits)

A parser for the prefix form

- The following grammar specifies the prefix form (Polish notation)

```
<exp> = <operand> | <op> <exp> <exp>
<operand> = float
<op>  = + | * | - | /
```

  - Example
    - (5.3 + 12.1) * 3.7 → * + 5.3 12.1 3.7



- Write a recursive program to implement this grammar

Implementation

```
int main(int argc, char *argv[]) {
  float result;
  int pos=0;

  if (argc < 2) {
    fprintf(stderr, "Error: missing parameter.\n");
    fprintf(stderr, "Run as: %s prefix_expression\n",
      argv[0]);
    return 1;
  }

  result = eval_r(argv[1], &pos);
  fprintf(stdout, "Result = %.2f\n", result);
  return EXIT_SUCCESS;
}
```

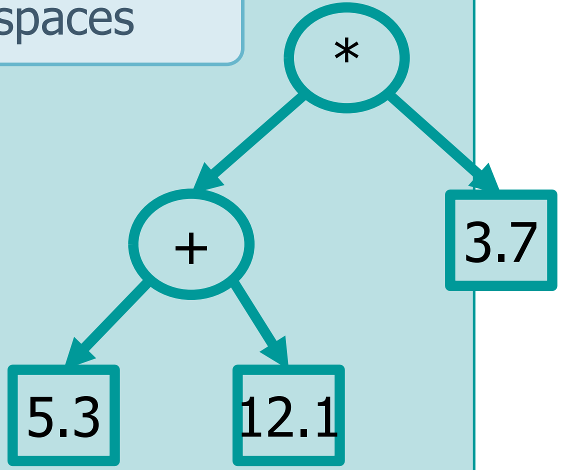
Implementation

```
float eval_r (char *expr, int *pos_ptr) {
  float left, right, result;
  char operator;
  int k = *pos_ptr;
  while (isspace(expr[k])) {
    k++;
  }
  if (expr[k]=='+' || expr[k]=='*' ||
      expr[k]=='-' || expr[k]=='/') {
    operator = expr[k++];
    left = eval_r(expr, &k);
    right = eval_r(expr, &k);
    switch (operator) {
      case '+': result = left+right; break;
      case '*': result = left*right; break;
      case '-': result = left-right; break;
      case '/': result = left/right; break;
    }
```

Expression

Parsing index

Skip spaces

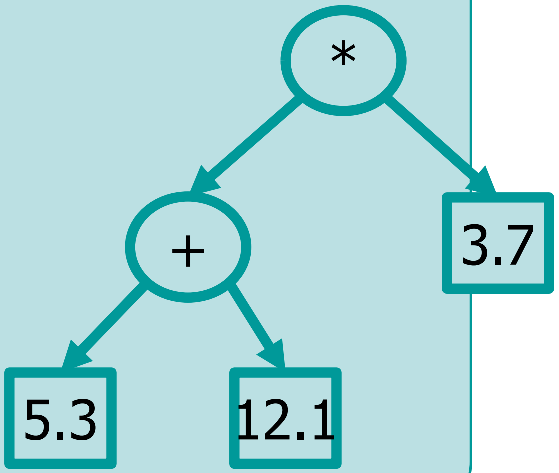


Implementation

```
  } else {
    sscanf(&expr[k], "%f", &result);
    while (isdigit(expr[k]) || expr[k]=='.') {
      k++;
    }
  }

  *pos_ptr = k;
  return result;
}
```

Terminal case:
A real value



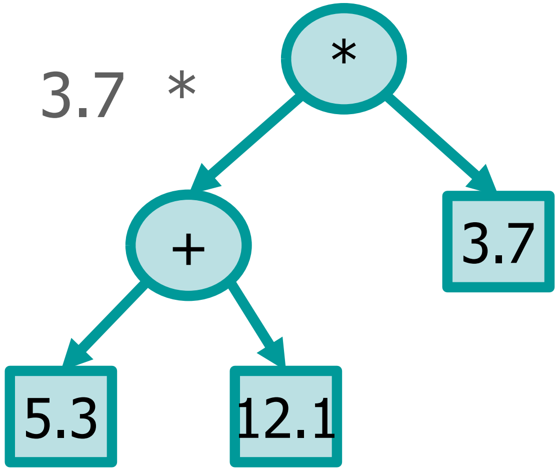
A parser for the postfix form

- The following grammar specifies the postfix form (Polish notation)

```
<exp> = <operand> | <exp> <exp> <op>
<operand> = float
<op>   = + | * | - | /
```

  - Example
    - (5.3 + 12.1) * 3.7 → 5.3 12.1 + 3.7 *

- Write a program to implement this grammar



Implementation

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#include "util.h"
#include "stackPublic.h"

int main(int argc, char *argv[]) {
  float result;
  int left, right, length, k=0;
  stack_t *sp=NULL;
  char *expr;

  util_check_m(argc>=2, "missing parameter.");
  expr = argv[1];
  length = strlen(expr);
  sp = stack_init(length);
```

ADT for utility functions

ADT for the stack

Implementation

```
while (k < length) {
  if (isdigit(expr[k])) {
    sscanf(&expr[k], "%f", &result);
    stack_push(sp, (void *)result);
    while (isdigit(expr[k]) || expr[k]=='.') {
      k++;
    }
} else if (expr[k]=='+' || expr[k]=='*' ||
    expr[k]=='-' || expr[k]=='/') {
    stack_pop(sp, (void **)&right);
    stack_pop(sp, (void **)&left);
    switch (expr[k]) {
      case '+': result = left+right; break;
      case '*': result = left*right; break;
      case '-': result = left-right; break;
      case '/': result = left/right; break;
    }
    stack_push(sp, (void *)result);
  }
  k++;
}
```

Skip float

Implementation

```
  stack_pop(sp, (void **)&result);
  fprintf(stdout, "Result = %ld\n", result);
  stack_dispose(sp, NULL);

  return EXIT_SUCCESS;
}
```