Lecture 15

- Segmented Array
- Advanced Parsing

Segmented Array

- Adding N elements in a doubling-array requires NlogN times of moving. (N elements times logN times resizing)
- The cost is not substantial if the cost for moving is small.
- Not the case if the cost for moving is not small.
- Or, what if the element is uncopyable and unmovable?
- Also there are situations that the address of the elements must not change.

Segmented Array

- It is like using a 2D array as a 1D array.
- Store a variable-length array of the pointers to the elements.
- Each pointer points to an array of the elements.
- When the array needs to grow, the variable-length array of the pointers is incremented and an array of elements is assigned.
- If the length of each array of elements is U, and the variable-length array of the pointers is ptrArray, the element at index=i can be accessed by:
  ptrArray[i/U][i%U]
- Since a computer can quickly calculate division by 2^N, U should be 2^N.

```
#ifndef SEGARRAY_IS_INCLUDED
#define SEGARRAY_IS_INCLUDED

#include <vector>

template <class T,const int bitShift>
class SegmentedArray
{
public:
    enum
    {
        UNIT_LENGTH=(1<<bitShift),
        MASK=(1<<bitShift)-1
        // Unit length=2^bitShift
    };

protected:
    long long int n;
    std::vector <T *> ptrArray;

public:
    SegmentedArray();
    ~SegmentedArray();
    void CleanUp(void);

    void Add(const T &incoming);
    long long int GetN(void) const;
    T &operator[](long long int idx);
    const T &operator[](long long int idx) const;
};

template <class T,const int bitShift>
SegmentedArray<T,bitShift>::SegmentedArray()
{
    n=0;
}
template <class T,const int bitShift>
SegmentedArray<T,bitShift>::~SegmentedArray()
{
    CleanUp();
}
template <class T,const int bitShift>
void SegmentedArray<T,bitShift>::CleanUp(void)
{
    n=0;
    for(auto &ptr : ptrArray)
    {
        delete [] ptr;
        ptr=nullptr;
    }
    ptrArray.clear();
}
template <class T,const int bitShift>
void SegmentedArray<T,bitShift>::Add(const T &incoming)
{
    auto majorIdx=(n>>bitShift);
    auto minorIdx=(n&MASK);

    if(ptrArray.size()<=majorIdx)
    {
        auto newPtr=new T [UNIT_LENGTH];
        ptrArray.push_back(newPtr);
    }

    ptrArray[majorIdx][minorIdx]=incoming;
    ++n;
}

template <class T,const int bitShift>
long long int SegmentedArray<T,bitShift>::GetN(void) const
{
    return n;
}
template <class T,const int bitShift>
T &SegmentedArray<T,bitShift>::operator[](long long int idx)
{
    auto majorIdx=(idx>>bitShift);
    auto minorIdx=(idx&MASK);
    return     ptrArray[majorIdx][minorIdx];
}
template <class T,const int bitShift>
const T &SegmentedArray<T,bitShift>::operator[](long long int idx) const
{
    auto majorIdx=(idx>>bitShift);
    auto minorIdx=(idx&MASK);
    return     ptrArray[majorIdx][minorIdx];
}

#endif
```

- Kind of using a 2D array as a 1D array.
- The biggest benefit is that the pointers of the elements do not change.

Advanced Parsing

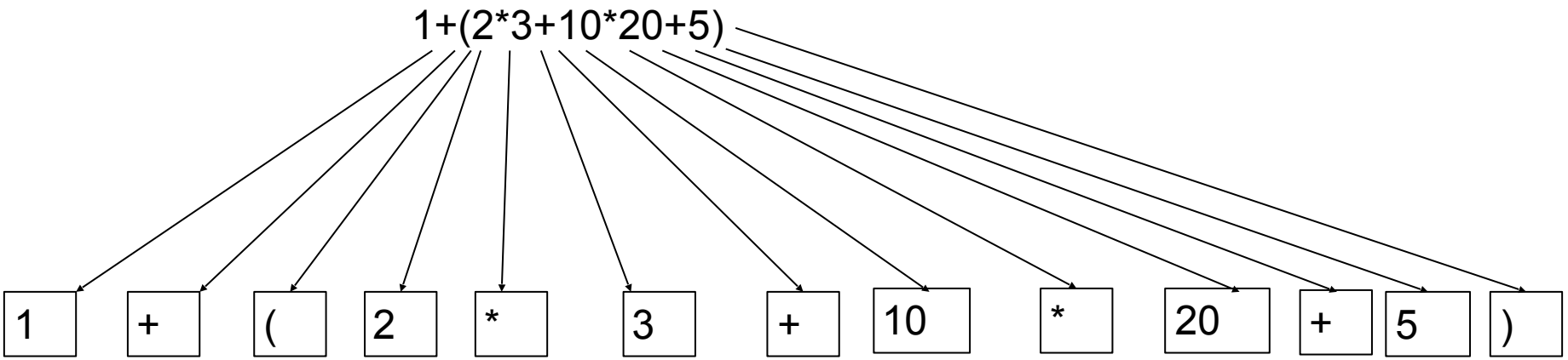
- How to recognize a string like:

  1+2*3

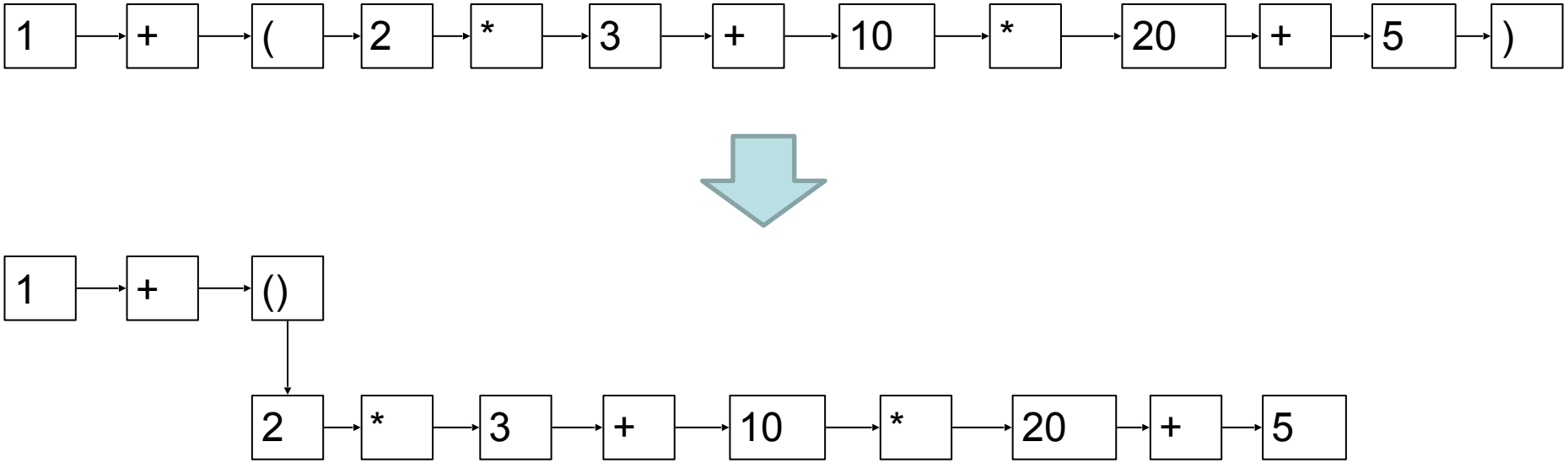
  1+(2*3+10*20)

Step 1 - Decomposition

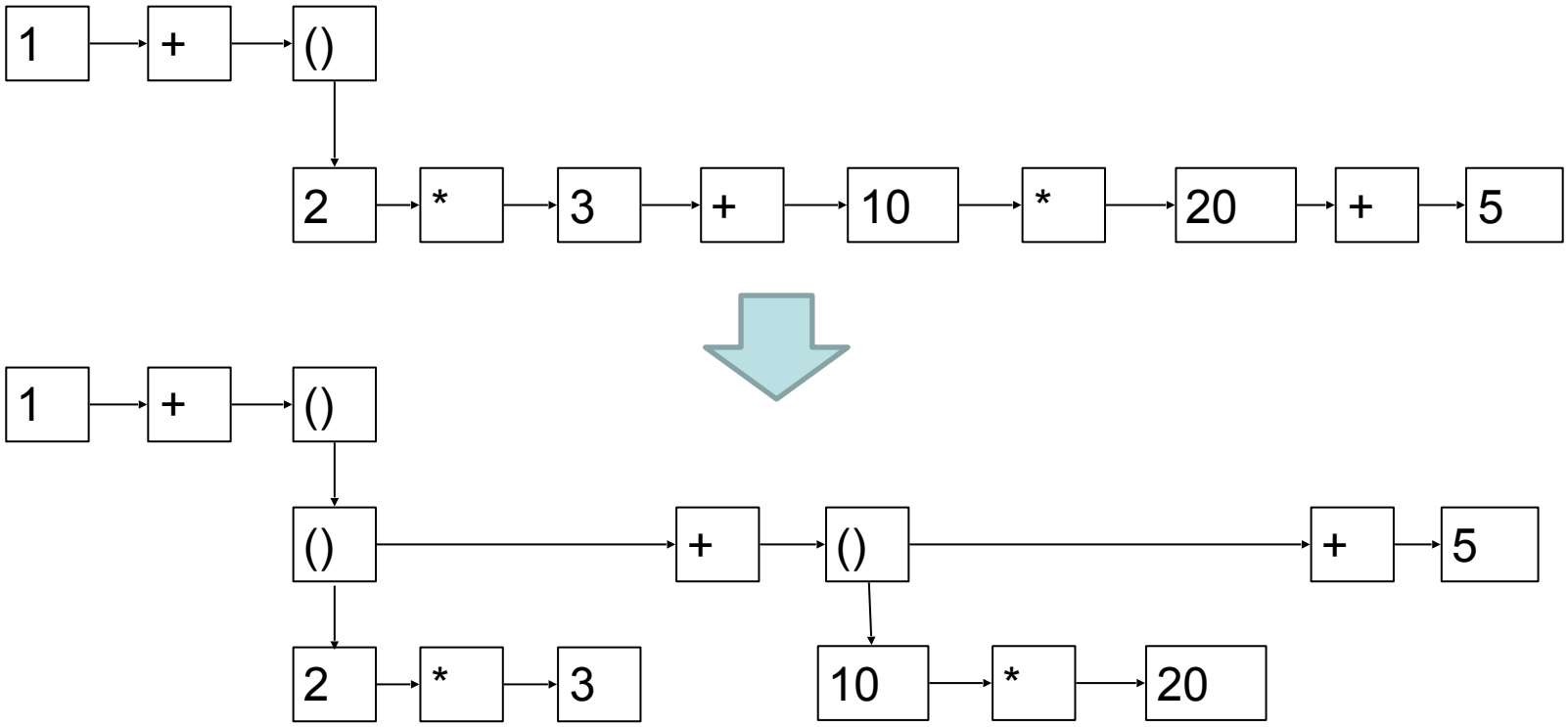
- A string needs to be decomposed into words and operators.



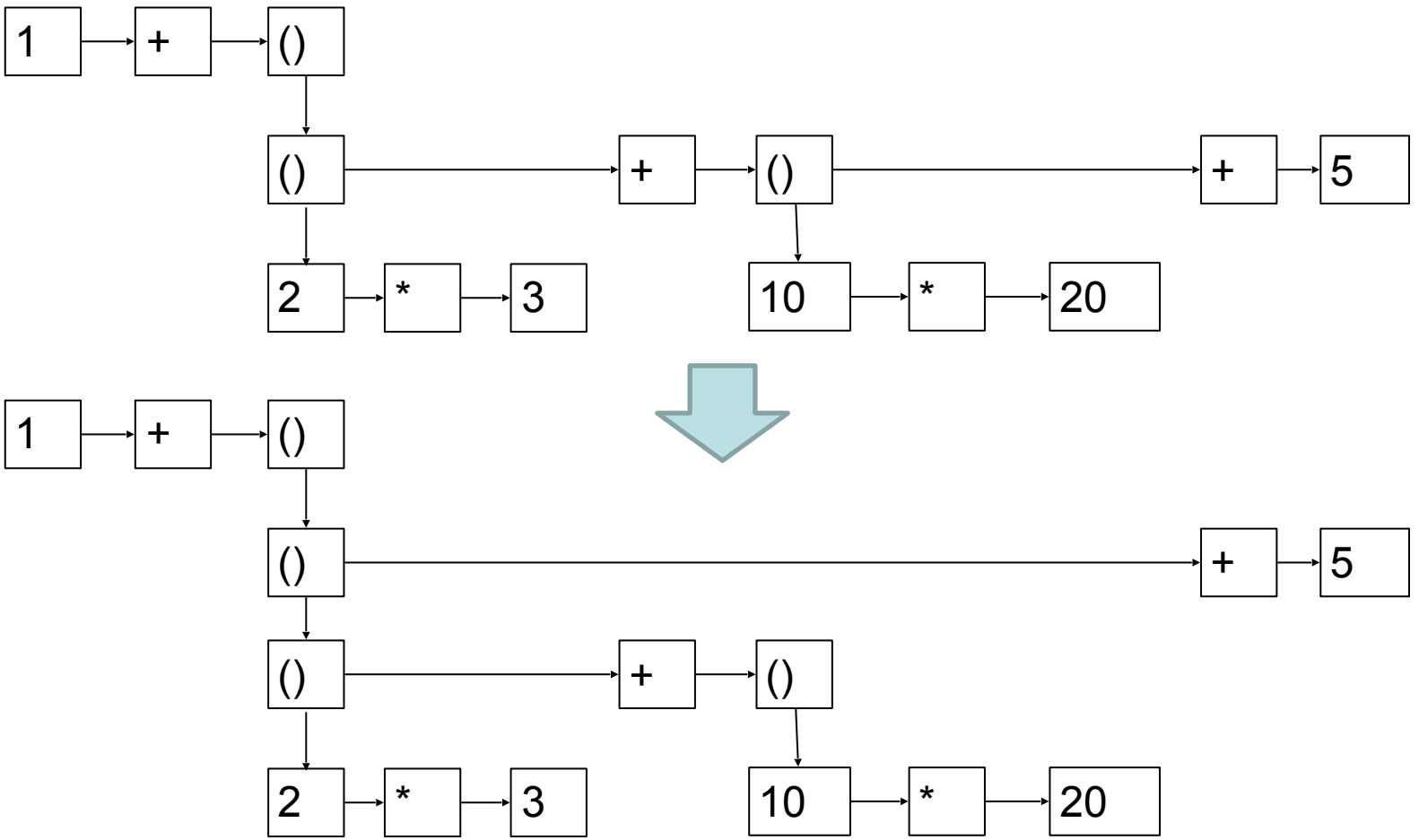
Step 2 – Group by parenthesis



Step 3 – Group by Higher Priority Operators

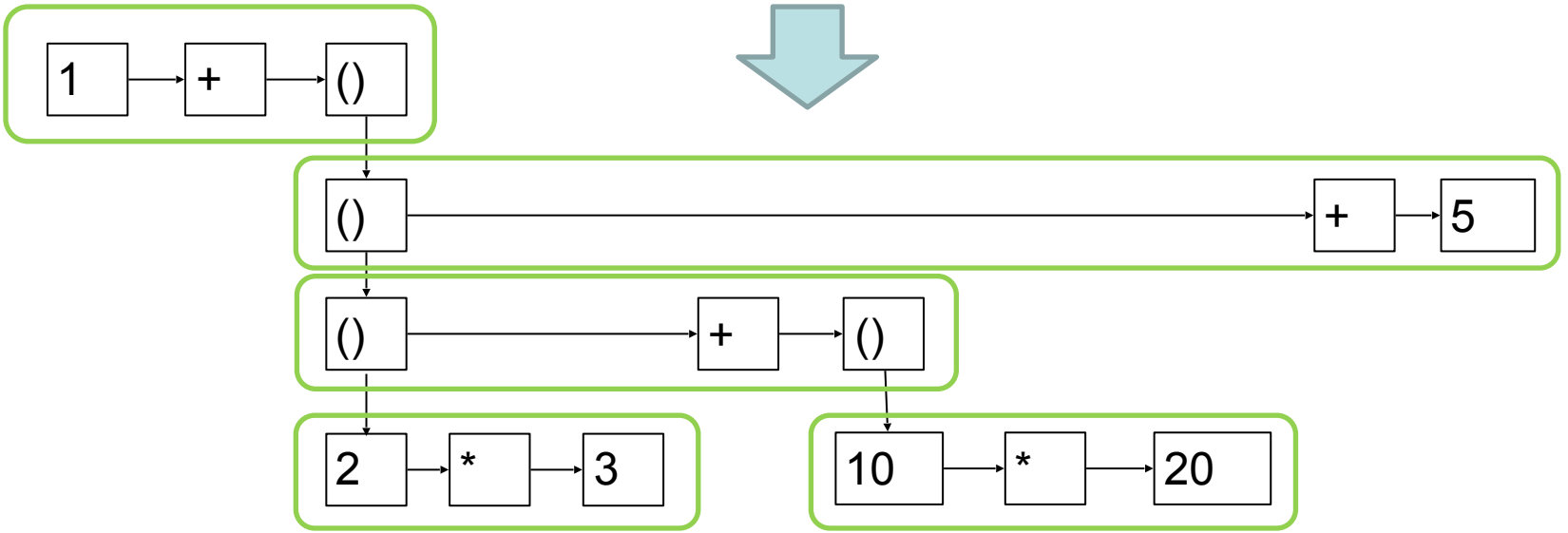


Step 4 – Group by Low-Priority Operators



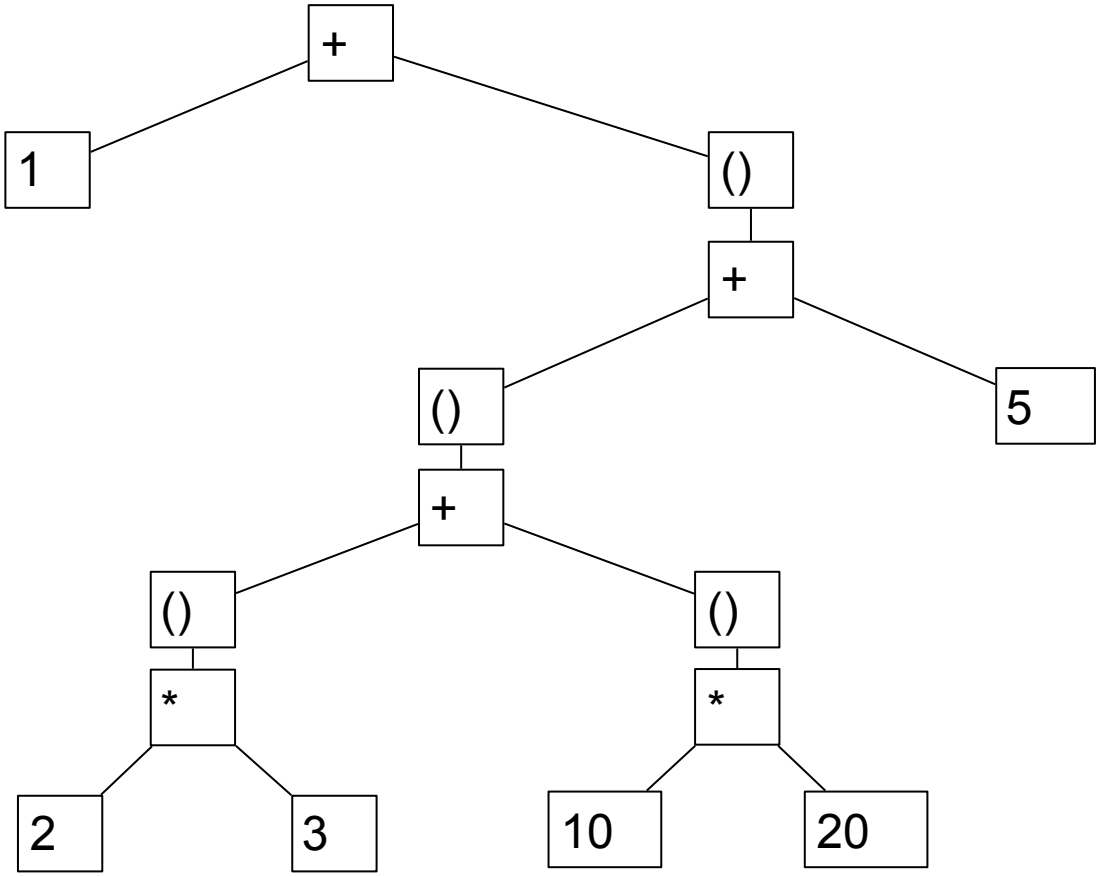
All Binary Operators

Finally, every group takes a form of a binary operator (a value + operator + a value)



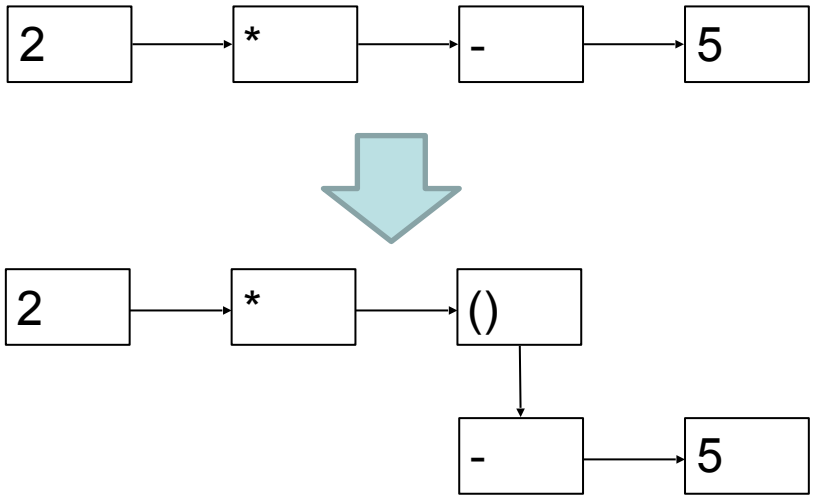
Optionally, can be made a binary-tree of the expression.

- I don’t do it today, but it is possible to reform it to a binary-tree expression.



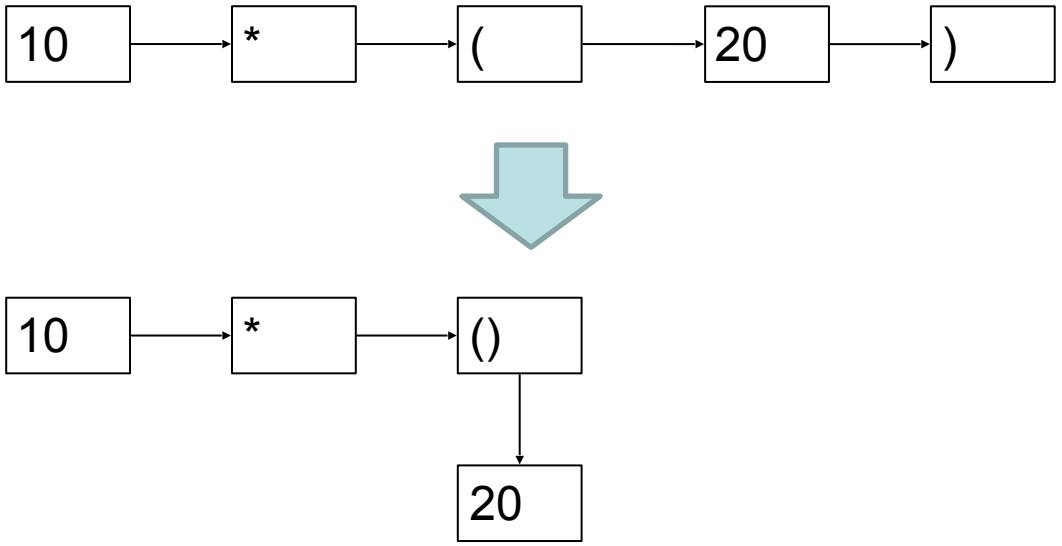
Wait, what about a unary operator?

- Need Step 2.5 – Group by Unary Operators
- The user may enter:
  2*-5 : Unary operator immediately after another operator
  -10+20 : Unary operator at the beginning
- After grouping by parenthesis, unary operators must be identified and grouped.



What about only one number in a parenthesis?

- The user may enter:
  10*(20)
- In this case, only one number will be in a group.

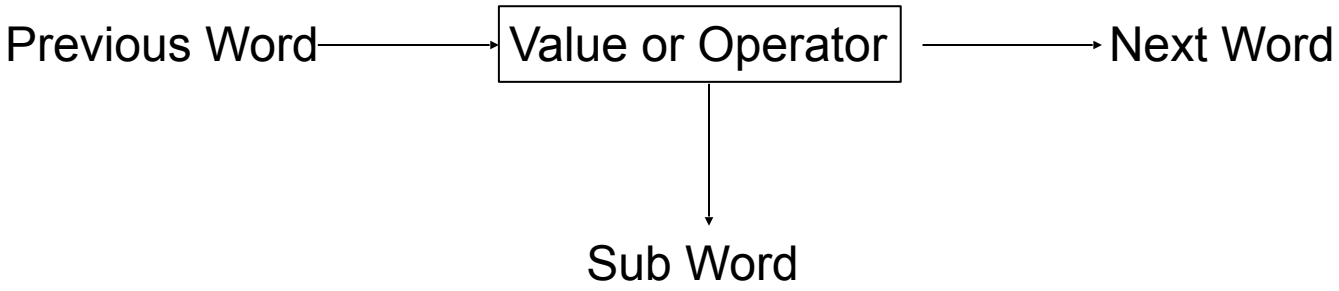


In the end…

- Every group must be:
  - A single value,
  - Two values separated by a binary operator, or
  - One value preceded by a unary operator.

Data structure

- Obviously, each word needs to have three links.



Step 1 Decomposition

```
#include <stdio.h>
#include <ysclass.h>


class Parser
{
public:
    class Word
    {
    public:
        YsWString str;
        Word *subWordPtr;
        Word *nextWordPtr;
        Word *prevWordPtr;
    };
    YsSegmentedArray <Word,4> wordArray;
    Word *firstWordPtr,*lastWordPtr;

public:
    Parser();
    ~Parser();
    void CleanUp(void);
    void Print(void);

    YSRESULT Parse(const YsWString &wstr);
protected:
    YSRESULT Decompose(const YsWString &wstr);
    Word *CreateWord(void);
};
```

```
Parser::Parser()
{
    firstWordPtr=nullptr;
    lastWordPtr=nullptr;
}
Parser::~Parser()
{
    CleanUp();
}
void Parser::CleanUp(void)
{
    wordArray.CleanUp();
    firstWordPtr=nullptr;
    lastWordPtr=nullptr;
}

void Parser::Print(void)
{
    for(auto ptr=firstWordPtr; nullptr!=ptr; ptr=ptr->nextWordPtr)
    {
        printf("%ls\n",ptr->str.Txt());
    }
}

YSRESULT Parser::Parse(const YsWString &wstr)
{
    CleanUp();
    if(YSOK!=Decompose(wstr))
    {
        return YSERR;
    }
    return YSOK;
}
```

```
YSRESULT Parser::Decompose(const YsWString &wstr)
{
    YsWString currentWord;
    for(YSSIZE_T idx=0; idx<wstr.Strlen(); ++idx)
    {
        if('+'==wstr[idx] ||
           '-'==wstr[idx] ||
           '*'==wstr[idx] ||
           '/'==wstr[idx] ||
           '%'==wstr[idx] ||
           '('==wstr[idx] ||
           ')'==wstr[idx] ||
           '['==wstr[idx] ||
           ']'==wstr[idx] ||
           '{'==wstr[idx] ||
           '}'==wstr[idx])
        {
            if(0<currentWord.Strlen())
            {
                auto newWord=CreateWord();
                newWord->str=currentWord;
                currentWord=L"";
            }
            auto newWord=CreateWord();
            newWord->str=L"";
            newWord->str.Append(wstr[idx]);
        }
        else
        {
            currentWord.Append(wstr[idx]);
        }
    }

    if(0<currentWord.Strlen())
    {
        auto newWord=CreateWord();
        newWord->str=currentWord;
    }

    return YSOK;
}
```

```
Parser::Word *Parser::CreateWord(void)
{
    auto nextIdx=wordArray.GetN();
    wordArray.Increment();

    auto newWordPtr=&wordArray[nextIdx];

    newWordPtr->prevWordPtr=lastWordPtr;
    if(nullptr!=lastWordPtr)
    {
        lastWordPtr->nextWordPtr=newWordPtr;
    }
    else
    {
        firstWordPtr=newWordPtr;
    }
    lastWordPtr=newWordPtr;
    newWordPtr->subWordPtr=nullptr;
    newWordPtr->nextWordPtr=nullptr;
    return &wordArray[nextIdx];
}


////////////////////////////////////////////////////////////



int main(int ac,char *av[])
{
    if(2<=ac)
    {
        YsWString wstr;
        wstr.SetUTF8String(av[1]);
        Parser parser;
        parser.Parse(wstr);
        parser.Print();
    }
    return 0;
}
```

Step 2 – Group by Parenthesis

- Changes in the Parser class definition:

```
class Parser
{
public:
    class Word
    {
    public:
        YsWString str;
        Word *subWordPtr;
        Word *nextWordPtr;
        Word *prevWordPtr;
    };
    YsSegmentedArray <Word,4> wordArray;
    Word *firstWordPtr,*lastWordPtr;

public:
    Parser();
    ~Parser();
    void CleanUp(void);
    void Print(void) const;
protected:
    void Print(const Word *wordPtr,YsString indent) const;

public:
    YSRESULT Parse(const YsWString &wstr);
protected:
    YSRESULT Decompose(const YsWString &wstr);
    YSRESULT ClampParenthesis(Word *&currentPtr,wchar_t leftSymbol);
    void DropClosingParenthesis(Word *currentPtr);
    Word *CreateWord(void);
};
```

- Changes in the Print function (Need to show hierarchical structure)

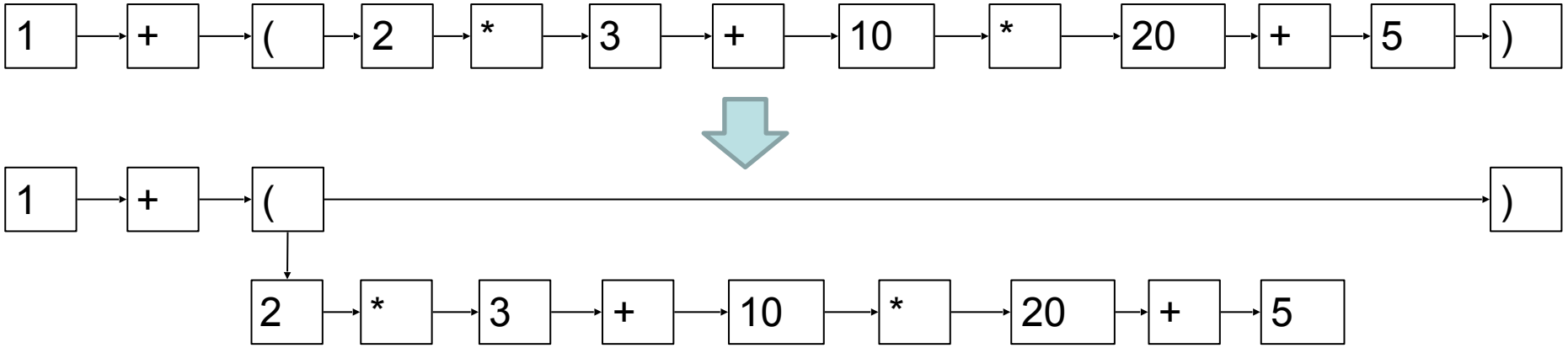
```
void Parser::Print(void) const
{
    Print(firstWordPtr,"");
}

void Parser::Print(const Word *wordPtr,YsString indent) const
{
    for(auto ptr=wordPtr; nullptr!=ptr; ptr=ptr->nextWordPtr)
    {
        printf("%s",indent.Txt());
        printf("%ls\n",ptr->str.Txt());
        if(nullptr!=ptr->subWordPtr)
        {
            auto newIndent=indent;
            newIndent.Append("  ");
            Print(ptr->subWordPtr,newIndent);
        }
    }
}
```

- Changes in the Parse function

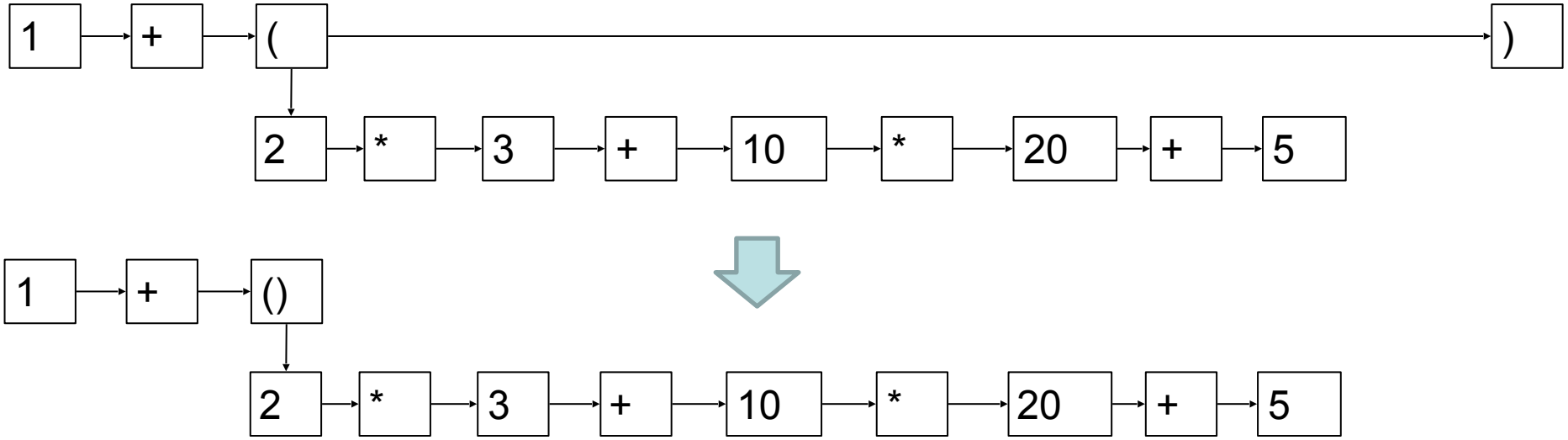
```
YSRESULT Parser::Parse(const YsWString &wstr)
{
    CleanUp();
    if(YSOK!=Decompose(wstr))
    {
        return YSERR;
    }

    auto top=firstWordPtr;
    if(YSOK!=ClampParenthesis(top,0))
    {
        return YSERR;
    }

    DropClosingParenthesis(firstWordPtr);

    return YSOK;
}
```

Two new functions

- ClampParenthesis:



- DropClosingParenthesis



```
YSRESULT Parser::ClampParenthesis(Word *&currentPtr,wchar_t leftSymbol)
{
   while(nullptr!=currentPtr)
   {
      if(0==currentPtr->str.Strcmp(L"(") || 0==currentPtr->str.Strcmp(L"[") ||
        0==currentPtr->str.Strcmp(L"{"))
      {
         auto subGroupPtr=currentPtr->nextWordPtr;
         if(YSOK!=ClampParenthesis(subGroupPtr,currentPtr->str[0]))
         {
            printf("Detected an open %ls\n",currentPtr->str.Txt());
            return YSERR;
         }
         subGroupPtr->prevWordPtr->nextWordPtr=nullptr;
         currentPtr->subWordPtr=currentPtr->nextWordPtr;

         currentPtr->nextWordPtr=subGroupPtr;
         subGroupPtr->prevWordPtr=currentPtr;

         currentPtr=subGroupPtr;
      }
      else if(0==currentPtr->str.Strcmp(L")") || 0==currentPtr->str.Strcmp(L"]") ||
            0==currentPtr->str.Strcmp(L"}"))
      {
         if((leftSymbol!='(' && currentPtr->str[0]==')') ||
           (leftSymbol!='[' && currentPtr->str[0]==']') ||
           (leftSymbol!='{' && currentPtr->str[0]=='}'))
         {
            printf("Miss-matching %ls\n",currentPtr->str.Txt());
            return YSERR;
         }
         return YSOK;
      }
      currentPtr=currentPtr->nextWordPtr;
   }
   if(0!=leftSymbol)
   {
      printf("Open %lc error.\n",leftSymbol);
      return YSERR;
   }
   return YSOK;
}
```

If successful, subGroupPtr points to the closing symbol for the currentPtr.

If closing, does it match the opening parenthesis?

```
void Parser::DropClosingParenthesis(Word *currentPtr)
{
   while(nullptr!=currentPtr)
   {
      DropClosingParenthesis(currentPtr->subWordPtr);

      if(nullptr!=currentPtr->nextWordPtr &&
         ((currentPtr->str[0]=='(' && currentPtr->nextWordPtr->str[0]==')') ||
          (currentPtr->str[0]=='[' && currentPtr->nextWordPtr->str[0]==']') ||
          (currentPtr->str[0]=='{' && currentPtr->nextWordPtr->str[0]=='}')))
      {
         currentPtr->str.Append(currentPtr->nextWordPtr->str);
         currentPtr->nextWordPtr=currentPtr->nextWordPtr->nextWordPtr;
         if(nullptr!=currentPtr->nextWordPtr)
         {
            currentPtr->nextWordPtr->prevWordPtr=currentPtr;
         }
      }

      currentPtr=currentPtr->nextWordPtr;
   }
}
```

Step 03 – Group Operators

- Changes in the Parser class definition.

```
class Parser
{
public:
    class Word
    {
    public:
        YsWString str;
        Word *subWordPtr;
        Word *nextWordPtr;
        Word *prevWordPtr;
    };
    YsSegmentedArray <Word,4> wordArray;
    Word *firstWordPtr,*lastWordPtr;

public:
    Parser();
    ~Parser();
    void CleanUp(void);
    void Print(void) const;
protected:
    void Print(const Word *wordPtr,YsString indent) const;

public:
    YSRESULT Parse(const YsWString &wstr);
protected:
    YSRESULT Decompose(const YsWString &wstr);
    YSRESULT ClampParenthesis(Word *&currentPtr,wchar_t leftSymbol);
    void DropClosingParenthesis(Word *currentPtr);
    void GroupOperator(const wchar_t * const op[]);
    void GroupOperator(Word *currentPtr,const wchar_t * const op[]);
    Word *CreateWord(void);
    Word *CreateWordNoConnection(void);
};
```

- Changes in the Parse function.

```
YSRESULT Parser::Parse(const YsWString &wstr)
{
    CleanUp();
    if(YSOK!=Decompose(wstr))
    {
        return YSERR;
    }

    auto top=firstWordPtr;
    if(YSOK!=ClampParenthesis(top,0))
    {
        return YSERR;
    }

    DropClosingParenthesis(firstWordPtr);

    const wchar_t * const mulDiv[]=
    {
        L"*",L"/",L"%",nullptr
    };
    GroupOperator(mulDiv);

    const wchar_t * const addSub[]=
    {
        L"+",L"-",nullptr
    };
    GroupOperator(addSub);

    return YSOK;
}
```

High-priority operators

Low-priority operators

```
void Parser::GroupOperator(const wchar_t * const op[])
{
   GroupOperator(firstWordPtr,op);
}

void Parser::GroupOperator(Word *currentPtr,const wchar_t * const op[])
{
   while(nullptr!=currentPtr)
   {
      GroupOperator(currentPtr->subWordPtr,op);

      if(nullptr!=currentPtr &&
        nullptr!=currentPtr->nextWordPtr &&
        nullptr!=currentPtr->nextWordPtr->nextWordPtr &&
        (nullptr!=currentPtr->nextWordPtr->nextWordPtr->nextWordPtr ||
         nullptr!=currentPtr->prevWordPtr))
      {
         auto operatorPtr=currentPtr->nextWordPtr;
         bool isOp=false;
         for(int i=0; nullptr!=op[i]; ++i)
         {
            if(0==operatorPtr->str.Strcmp(op[i]))
            {
               isOp=true;
               break;
            }
         }
```

Value

Operator

Value

If there is no word before and after, it is already two values separated with a binary operator.

This transformation is needed only when there is a word before or after.

Is it really the operator I am looking for?

(Continued)

Value Operator Value

```
        if(isOp)
        {
            auto afterTerm=currentPtr->nextWordPtr->nextWordPtr->nextWordPtr;

            auto subWordPtr=CreateWordNoConnection();
            subWordPtr->str=(YsWString &&)currentPtr->str;
            subWordPtr->subWordPtr=currentPtr->subWordPtr;
            currentPtr->str=L"()";

            currentPtr->subWordPtr=subWordPtr;
            subWordPtr->nextWordPtr=currentPtr->nextWordPtr;
            subWordPtr->nextWordPtr->prevWordPtr=subWordPtr;

            subWordPtr->nextWordPtr->nextWordPtr->nextWordPtr=nullptr;

            currentPtr->nextWordPtr=afterTerm;
            if(nullptr!=afterTerm)
            {
                afterTerm->prevWordPtr=currentPtr;
            }
        }
        else
        {
            currentPtr=currentPtr->nextWordPtr;
        }
    }
    else
    {
        currentPtr=currentPtr->nextWordPtr;
    }
  }
}
```

Bunch of disconnections and re-connections.

```
Parser::Word *Parser::CreateWordNoConnection(void)
{
    auto nextIdx=wordArray.GetN();
    wordArray.Increment();

    auto newWordPtr=&wordArray[nextIdx];
    newWordPtr->prevWordPtr=nullptr;
    newWordPtr->nextWordPtr=nullptr;
    newWordPtr->subWordPtr=nullptr;

    return newWordPtr;
}
```

Step 4 – Identify and Group Unary Operators

```
class Parser
{
public:
    class Word
    {
    public:
        YsWString str;
        Word *subWordPtr;
        Word *nextWordPtr;
        Word *prevWordPtr;
    };
    YsSegmentedArray <Word,4> wordArray;
    Word *firstWordPtr,*lastWordPtr;

public:
    Parser();
    ~Parser();
    void CleanUp(void);
    void Print(void) const;
protected:
    void Print(const Word *wordPtr,YsString indent) const;

public:
    YSRESULT Parse(const YsWString &wstr);
protected:
    YSRESULT Decompose(const YsWString &wstr);
    YSRESULT ClampParenthesis(Word *&currentPtr,wchar_t leftSymbol);
    void DropClosingParenthesis(Word *currentPtr);
    void GroupUnaryOperator(Word *currentPtr,const wchar_t *const allOp[],const wchar_t *const unaryOp[]);
    void GroupOperator(const wchar_t * const op[]);
    void GroupOperator(Word *currentPtr,const wchar_t * const op[]);
    Word *CreateWord(void);
    Word *CreateWordNoConnection(void);
};
```

```
YSRESULT Parser::Parse(const YsWString &wstr)
{
    CleanUp();
    if(YSOK!=Decompose(wstr))
    {
        return YSERR;
    }

    auto top=firstWordPtr;
    if(YSOK!=ClampParenthesis(top,0))
    {
        return YSERR;
    }

    DropClosingParenthesis(firstWordPtr);

    const wchar_t * const allOp[]=
    {
        L"+",L"-",L"*",L"/",L"%",nullptr
    };
    const wchar_t * const unaryOp[]=
    {
        L"+",L"-",nullptr
    };
    GroupUnaryOperator(firstWordPtr,allOp,unaryOp);

    const wchar_t * const mulDiv[]=
    {
        L"*",L"/",L"%",nullptr
    };
    GroupOperator(mulDiv);

    const wchar_t * const addSub[]=
    {
        L"+",L"-",nullptr
    };
    GroupOperator(addSub);

    return YSOK;
}
```

Identify a can-be unary operator after another operator.

```
void Parser::GroupUnaryOperator(Word *currentPtr,const wchar_t *const allOp[],const wchar_t * const unaryOp[])
{
    while(nullptr!=currentPtr)
    {
        bool prevIsOp=false;
        if(nullptr==currentPtr->prevWordPtr)
        {
            prevIsOp=true;
        }
        else
        {
            for(int i=0; nullptr!=allOp[i]; ++i)
            {
                if(0==currentPtr->prevWordPtr->str.Strcmp(allOp[i]))
                {
                    prevIsOp=true;
                    break;
                }
            }
        }

        bool currentIsUnary=false;
        for(int i=0; nullptr!=unaryOp[i]; ++i)
        {
            if(0==currentPtr->str.Strcmp(unaryOp[i]))
            {
                currentIsUnary=true;
                break;
            }
        }
```

(Continued)

```
        if(prevIsOp && currentIsUnary && nullptr!=currentPtr->nextWordPtr)
        {
            auto newNextWordPtr=currentPtr->nextWordPtr->nextWordPtr;

            auto subWordPtr=CreateWordNoConnection();

            subWordPtr->str=(YsWString &&)currentPtr->str;
            subWordPtr->subWordPtr=currentPtr->subWordPtr; // Not supposed to have one, but just in case.

            subWordPtr->nextWordPtr=currentPtr->nextWordPtr;
            subWordPtr->nextWordPtr->prevWordPtr=subWordPtr;

            currentPtr->str=L"()";
            currentPtr->subWordPtr=subWordPtr;

            subWordPtr->prevWordPtr=nullptr;
            subWordPtr->nextWordPtr->nextWordPtr=nullptr;

            currentPtr->nextWordPtr=newNextWordPtr;
            if(nullptr!=newNextWordPtr)
            {
                newNextWordPtr->prevWordPtr=currentPtr;
            }
        }
        else
        {
            currentPtr=currentPtr->nextWordPtr;
        }
    }
}
```

Step 5 - Evaluate

```
class Parser
{
public:
    class Word
    {
    public:
        YsWString str;
        Word *subWordPtr;
        Word *nextWordPtr;
        Word *prevWordPtr;
    };
    YsSegmentedArray <Word,4> wordArray;
    Word *firstWordPtr,*lastWordPtr;

public:
    Parser();
    ~Parser();
    void CleanUp(void);
    void Print(void) const;
protected:
    void Print(const Word *wordPtr,YsString indent) const;

public:
    YSRESULT Parse(const YsWString &wstr);
protected:
    YSRESULT Decompose(const YsWString &wstr);
    YSRESULT ClampParenthesis(Word *&currentPtr,wchar_t leftSymbol);
    void DropClosingParenthesis(Word *currentPtr);
    void GroupUnaryOperator(Word *currentPtr,const wchar_t *const allOp[],const wchar_t *const unaryOp[]);
    void GroupOperator(const wchar_t * const op[]);
    void GroupOperator(Word *currentPtr,const wchar_t * const op[]);
    Word *CreateWord(void);
    Word *CreateWordNoConnection(void);

public:
    double EvaluateAsDouble(void) const;
protected:
    double EvaluateAsDouble(const Word *currentPtr) const;
    double EvaluateSingleNodeAsDouble(const Word *currentPtr) const;
};
```

```
double Parser::EvaluateAsDouble(void) const
{
   return EvaluateAsDouble(firstWordPtr);
}

double Parser::EvaluateAsDouble(const Word *currentPtr) const
{
   if(nullptr!=currentPtr && nullptr==currentPtr->nextWordPtr)
   {
      return EvaluateSingleNodeAsDouble(currentPtr);
   }

   if(nullptr!=currentPtr &&
     nullptr!=currentPtr->nextWordPtr &&
     nullptr==currentPtr->nextWordPtr->nextWordPtr)
   {
      // Can be a unary operator
      if(0==currentPtr->str.Strcmp(L"+"))
      {
         return EvaluateSingleNodeAsDouble(currentPtr->nextWordPtr);
      }
      else if(0==currentPtr->str.Strcmp(L"-"))
      {
         return -EvaluateSingleNodeAsDouble(currentPtr->nextWordPtr);
      }
      printf("Unexpected unary operator %ls\n",currentPtr->str.Txt());
      return 0.0;
   }
```

Single word

Unary Operator

```
    if(nullptr!=currentPtr &&
       nullptr!=currentPtr->nextWordPtr &&
       nullptr!=currentPtr->nextWordPtr->nextWordPtr &&
       nullptr==currentPtr->nextWordPtr->nextWordPtr->nextWordPtr)
    {
        // Can be a binary operator
        double left=EvaluateSingleNodeAsDouble(currentPtr);
        double right=EvaluateSingleNodeAsDouble(currentPtr->nextWordPtr->nextWordPtr);

        if(0==currentPtr->nextWordPtr->str.Strcmp(L"+"))
        {
            return left+right;
        }
        else if(0==currentPtr->nextWordPtr->str.Strcmp(L"-"))
        {
            return left-right;
        }
        else if(0==currentPtr->nextWordPtr->str.Strcmp(L"*"))
        {
            return left*right;
        }
        else if(0==currentPtr->nextWordPtr->str.Strcmp(L"/"))
        {
            return left/right;
        }
        else if(0==currentPtr->nextWordPtr->str.Strcmp(L"%"))
        {
            return fmod(left,right);
        }

        printf("Unrecognized operator. %ls\n",currentPtr->nextWordPtr->str.Txt());
        return 0.0;
    }

    printf("Unexpected pattern error.\n");
    Print(currentPtr,"ERROR:");
    return 0.0;
}
```

Binary operator

```
double Parser::EvaluateSingleNodeAsDouble(const Word *currentPtr) const
{
    if(nullptr!=currentPtr->subWordPtr)
    {
        return EvaluateAsDouble(currentPtr->subWordPtr);
    }

    YsString str;
    str.EncodeUTF8 <wchar_t> (currentPtr->str);
    return atof(str);
}
```

This example uses YsWString, which is a string of wchar_t. Needs to be converted to an 8-bit string for atof.

How about functions?

```
class Parser
{
public:
    class Word
    {
    public:
        YsWString str;
        Word *subWordPtr;
        Word *nextWordPtr;
        Word *prevWordPtr;
    };
    YsSegmentedArray <Word,4> wordArray;
    Word *firstWordPtr,*lastWordPtr;

public:
    Parser();
    ~Parser();
    void CleanUp(void);
    void Print(void) const;
protected:
    void Print(const Word *wordPtr,YsString indent) const;

public:
    YSRESULT Parse(const YsWString &wstr);
protected:
    YSRESULT Decompose(const YsWString &wstr);
    YSRESULT ClampParenthesis(Word *&currentPtr,wchar_t leftSymbol);
    void DropClosingParenthesis(Word *currentPtr);
    void GroupFunction(Word *currentPtr,const wchar_t * const func[]);
    void GroupUnaryOperator(Word *currentPtr,const wchar_t *const allOp[],const wchar_t *const unaryOp[]);
    void GroupOperator(const wchar_t * const op[]);
    void GroupOperator(Word *currentPtr,const wchar_t * const op[]);
    Word *CreateWord(void);
    Word *CreateWordNoConnection(void);

public:
    double EvaluateAsDouble(void) const;
protected:
    double EvaluateAsDouble(const Word *currentPtr) const;
    double EvaluateSingleNodeAsDouble(const Word *currentPtr) const;
};
```

```
YSRESULT Parser::Parse(const YsWString &wstr)
{
    CleanUp();
    if(YSOK!=Decompose(wstr))
    {
        return YSERR;
    }

    auto top=firstWordPtr;
    if(YSOK!=ClampParenthesis(top,0))
    {
        return YSERR;
    }

    DropClosingParenthesis(firstWordPtr);

    const wchar_t * const funcLabel[]=
    {
        L"sin",L"cos",L"tan",nullptr
    };
    GroupFunction(firstWordPtr,funcLabel);

    const wchar_t * const allOp[]=
    {
        L"+",L"-",L"*",L"/",L"%",nullptr
    };
    const wchar_t * const unaryOp[]=
    {
        L"+",L"-",nullptr
    };
    GroupUnaryOperator(firstWordPtr,allOp,unaryOp);

    const wchar_t * const mulDiv[]=
    {
        L"*",L"/",L"%",nullptr
    };
    GroupOperator(mulDiv);

    const wchar_t * const addSub[]=
    {
        L"+",L"-",nullptr
    };
    GroupOperator(addSub);

    return YSOK;
}
```

```
void Parser::GroupFunction(Word *currentPtr,const wchar_t * const func[])
{
    while(nullptr!=currentPtr)
    {
        GroupFunction(currentPtr->subWordPtr,func);

        bool isFunc=false;
        for(int i=0; nullptr!=func[i]; ++i)
        {
            if(0==currentPtr->str.Strcmp(func[i]))
            {
                isFunc=true;
                break;
            }
        }

        if(isFunc && nullptr!=currentPtr->nextWordPtr)
        {
            auto newNextWordPtr=currentPtr->nextWordPtr->nextWordPtr;

            currentPtr->subWordPtr=currentPtr->nextWordPtr;

            currentPtr->subWordPtr->prevWordPtr=nullptr;
            currentPtr->subWordPtr->nextWordPtr=nullptr;

            currentPtr->nextWordPtr=newNextWordPtr;
            if(nullptr!=newNextWordPtr)
            {
                newNextWordPtr->prevWordPtr=currentPtr;
            }
        }

        currentPtr=currentPtr->nextWordPtr;
    }
}
```

```
double Parser::EvaluateSingleNodeAsDouble(const Word *currentPtr) const
{
    if(0==currentPtr->str.Strcmp(L"sin") && nullptr!=currentPtr->subWordPtr)
    {
        return sin(EvaluateAsDouble(currentPtr->subWordPtr));
    }
    if(0==currentPtr->str.Strcmp(L"cos") && nullptr!=currentPtr->subWordPtr)
    {
        return cos(EvaluateAsDouble(currentPtr->subWordPtr));
    }
    if(0==currentPtr->str.Strcmp(L"tan") && nullptr!=currentPtr->subWordPtr)
    {
        return tan(EvaluateAsDouble(currentPtr->subWordPtr));
    }

    if(nullptr!=currentPtr->subWordPtr)
    {
        return EvaluateAsDouble(currentPtr->subWordPtr);
    }

    if(0==currentPtr->str.STRCMP(L"PI"))
    {
        return YsPi;
    }

    YsString str;
    str.EncodeUTF8 <wchar_t> (currentPtr->str);
    return atof(str);
}
```