Teilaufgabe 3: SYNTAX

Syntax

Wie sieht ein (syntaktisch) wohlgeformter arithmetischer Ausdruck aus?

Beispiele

```
1, 2, ...

1 + 2

(1 + 3) * 2

...
```

Formalere Sicht.

E ist ein arithmetischer Ausdruck genau dann wenn (gdw)

- E ist eine Zahl
- E ist ein zusammengesetzter Ausruck E1+E2, E1*E2 wobei jeweils E1, E2 arithmetische Ausdrücke sind
- E ist ein geklammerter Ausdruck (E1) wbei E1 ein aritmetischer Ausdruck ist.

Aha, die Regeln zur Beschreibung von arithmetischen Ausdrücken entsprechen Regel einer kontext-freien Grammatik

```
E -> 0
E -> 1
...
...
E -> E + E
E -> E * E
E -> (E)
```

Hm, Regeln aufwändig für Zahlen allgemein

```
E -> N
N -> 0
...
N -> 9
N -> 0N
...
N -> 9N
```

Kann stark vereinfacht werden mit Hilfe von regulären Ausdrücken! Z.B.

```
E -> ['0'-'9']+
```

wobei `r+ = r r*`. Sprich, einmalige Wiederholung von `r` gefolgt von `r*`. Wir schreiben `['0'-'9']` als Kurznotation für `'0'|...|'9'`.

Es gibt eine Reihen von Metasyntax Notationen zur Beschreibung von kontext-freien Sprachen. Siehe z.B. Extended Backus–Naur form.

Unsere Syntax von arithmetischen Ausdrücken

Angelehnt an EBNF, betrachten wir folgende Syntax. Der Einfachheithalber betrachten wir nur die Zahlen, 0, 1 und 2.

```
N ::= 0 | 1 | 2

E ::= N | (E) | E + E | E * E
```

Wir bezeichnen N und E als Nicht-Terminal Symbole und 0, 1, 2, (,), + und * als Terminal Symbole.

Was fällt auf? Die EBNF Grammatik ist nicht eindeutig!. D.h. für ein Wort gibt es zwei verschiedene Ableitungen.

```
(1)
        E
    -> E + E
    -> N + E
    -> 1 + E
    -> 1 + E * E
    -> 1 + N * E
    -> 1 + 2 * E
    -> 1 + 2 * N
    -> 1 + 2 * 1
```

```
(2)

        E
    -> E * E
    -> E + E * E
    -> 1 + E * E
    -> 1 + 2 * E
    -> 1 + 2 * 1
```

Was ist die Konsequenz verschiedener Ableitungen? Im Fall von (1) wird das Wort "1 + 2 * 1" interpretiert als 1 + (2 * 1). Im Fall von (2) wird das Wort "1 + 2 * 1" interpretiert als (1 + 2) * 1.

Eindeutige Grammatik (bzw. Regeln welche Nicht-eindeutigkeiten Einschränkungen) sind wichtig. Weil sonst könnte Compiler A (Oracle) die Interpretation (1) verwenden und Compiler B (Microsoft) die Interpretation

(2). Deshalb sollte wir eine eindeutige Beschreibung (der Syntax) von arithmetischen Ausdrücken haben.

In der Regel gehen wir von "Punkt vor Strich" aus. Gibt es eine äquivalente (eindeutige) Grammatik welche die "Punkt vor Strich" umsetzt? Ja. Betrachte folgende Grammatik.

```
E ::= E + T | T

T ::= T * F | F

F ::= N | (E)
```

Betrachte Beispiel von oben.

```
        E
    -> E + T
    -> E + T * F
    -> T + T * F
    -> F + T * F
    -> F + F * F
    -> N + F * F
    -> 1 + F * F
    -> 1 + N * F
    -> 1 + 2 * F
    -> 1 + 2 * N
    -> 1 + 2 * 1
```

Interpretiert als `1 + (2 * 1)`.

Darstellung in C++

Wie kann die Syntax in C++ dargestellt werden?

Idee:

- Basisklasse Expressions

- Abgeleitete Klassen für die einzelnen Fälle wie `IntExp`, `PlusExp` und `MultExp`.

Unten finden Sie eine Skizze. Beachte. Die Skizze ist stark vereinfacht im Vergleich zur Implementierung die Ihnen zur Verfügung gestellt ist.

```
class Exp {};

class IntExp : Exp {
   int val;
   IntExp(int x) { val = x; }
};

class PlusExp : Exp {
   Exp l,r;
   PlusExp(Exp x,y) { l = x; r = y; }
};

class MultExp : Exp {
   Exp l,r;
   MultExp(Exp x,y) { l = x; r = y; }
};
```

Beachte. Klammerung muss nicht in C++ dargestellt werden. Implizit behandelt durch Klammerung der jeweiligen Objekte.

Beispiel.

```
"1 + (3 + 5)"

==>

PlusExp(IntExp(1), PlusExp(IntExp(3), IntExp(5)))
```

Aha! In der C++ Darstellung ist die "Klammerung" implizit garantiert durch die Struktur des Objekts!

Konkrete versus Abstrakte Syntax

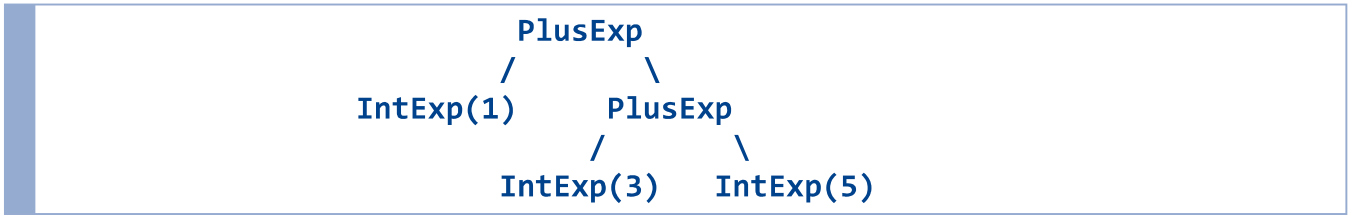
Die EBNF Darstellung der Grammatik wird als **konkrete Syntax** bezeichnet. Dies ist die Syntax wie sie die Programmerin sieht.

Die C++ Darstellung der Grammatik wird als **abstrakte Syntax** bezeichnet. Dies ist die Syntax von der Maschine verwendet, zur weiteren Verarbeitung des Programms.

Das Objekt

```
PlusExp(IntExp(1), PlusExp(IntExp(3), IntExp(5)))
```

stellt einen **abstrakten Syntax Baum** dar. Wir betrachten Aufrufe des Konstruktors `IntExp` als Blätter und Aufrufe von `PlusExp` und `MultExp` als (Zwischen)knoten, wobei Argumente Kinderknoten dartellen. Z.B.



Im Englischen wird der abstrake Syntax Baum als "abstract syntax tree (AST)" bezeichnet. Wir verwenden daher oft die Abkürzung AST wenn wir auf den abstrakten Syntax Baum uns beziehen.

Syntax Analyse (Parser/Parsing)

Parsing beschreibt den Vorgang welcher

- erkennt, ob ein Wort Teil der konkreten Syntax (Grammatik) ist, *und*
- (im Erfolgsfall) einen abstrakten Syntaxbaum liefert.

Ein Parser wird Ihnen zur Verfügung gestellt. Dessen Funktionsweise wird in der Vorlesung im Detail besprochen.

Teilaufgabe SYNTAX

Der ihnen zur Verfügung gestellte Programmcode kommt mit einem "pretty printer" welcher den AST in eine String umwandelt. Unser "pretty printer" ist sehr einfach gestrickt. Z.B. pretty-printing von

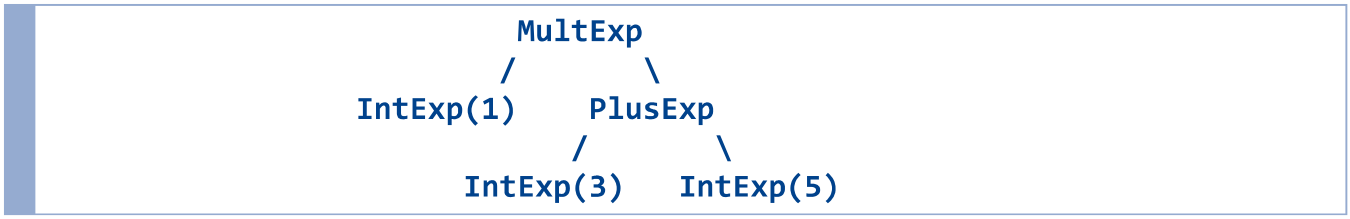
```
                PlusExp
               /       \
        IntExp(1)    PlusExp
                    /       \
              IntExp(3)   IntExp(5)
```

liefert den String

```
(1 + (3 + 5))
```

Sprich, jeder Ausdruck bestehend aus einem binären Operator wird geklammert. Durch die Klammerung wird gesichert, dass sich die Bedeutung des Ausdrucks nicht ändert.

Betrachte den AST



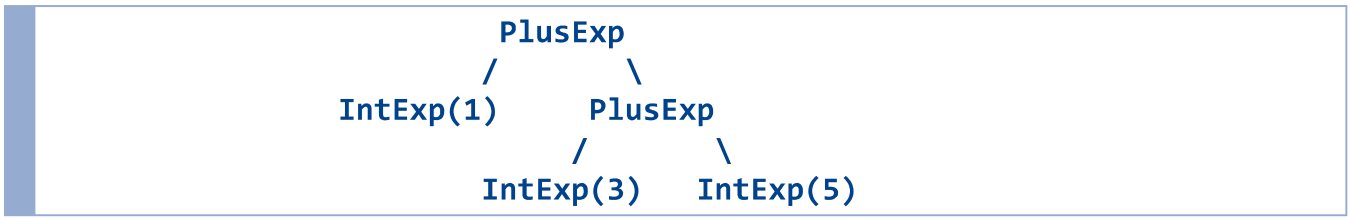
für welchen pretty printing folgenden String liefert.

```
(1 * (3 + 5))
```

Durch die Klammern wird der Ausdruck aber schwer(er) lesbar. Ziel dieser Teilaufgabe ist es einen "cleveren" pretty printer zu bauen, welcher soviele Klammern wie möglich eliminert.

Es folgende Beispiele, welche ihre clevere pretty print Variant handhaben sollte.

Für den AST

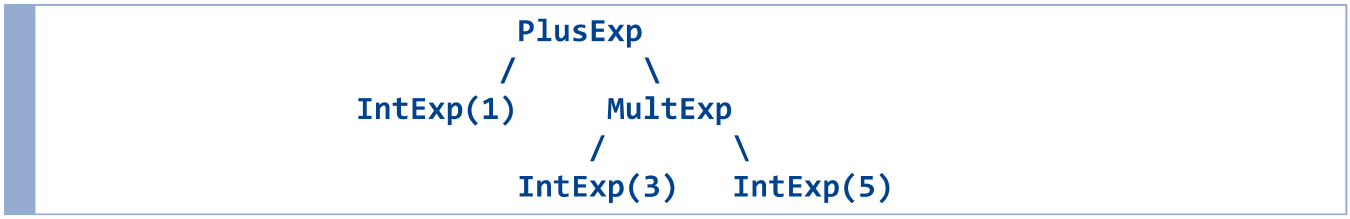


die clevere pretty print Variante, sollte folgenden String liefern

```
1 + 3 + 5
```

Da + assoziativ is eliminiere alle Klammern.

Für den AST

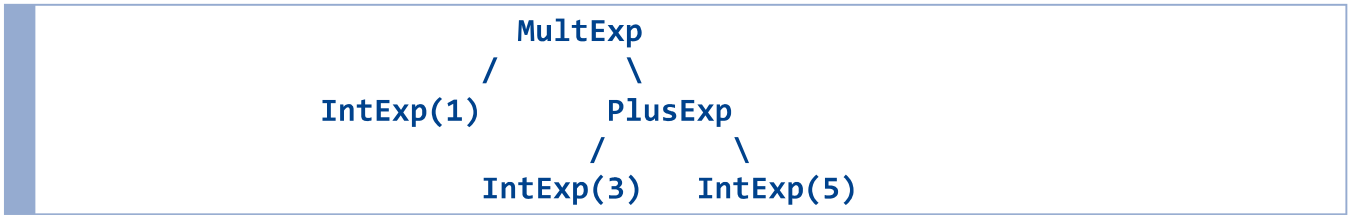


die clevere pretty print Variante, sollte folgenden String liefern

```
1 + 3 * 5
```

Wir gehen von "Punkt vor Strich" aus. Deshalb eliminiere alle Klammern.

Für den AST



die clevere pretty print Variante, sollte folgenden String liefern

```
1 * (3 + 5)
```

Die Klammer `(3 + 5)` kann nicht elminiert werden, da sich sonst die Bedeutung des Ausdrucks ändert.

Hinweis: Wie können Sie ihre Lösung testen? Parsen der "naiven" (gegebenen) pretty print Variant und ihrer "cleveren" Variante, sollte den gleichen AST liefern.

Beachte, der AST mag nicht notwendigerweise syntaktisch gleich sein.

Betrachte folgendes Beispiel.

```
Ausdruck

1 + (2 + 3)


liefert den AST

    PlusExp
    /     \
   1      PlusExp
          /    \
         2      3

Die clevere pretty Variante liefert den Ausdruck

1 + 2 + 3


Dieser (geparsed) ergibt den AST

         PlusExp
         /     \
     PlusExp    3
     /     \
     1     2


Die ASTs sind syntaktisch verschieden.
Unter der Annahme, dass Addition (+) assoziativ ist,
sind beide ASTs identisch.
```

Folgerungen.

1. In der Vergleichsroutine zweier ASTs wird erst normalisiert, z.B. alle Additionen Linksassoziativ dargestellt.

2. Anstatt dem AST Vergleich, werden beide ASTs interpretiert (eval). Das Ergebnis muss dann gleich sein.

Teilaufgabe 3: SEMANTIK

Wir betrachten die semantische Auswertung von Ausdrücken. Dazu gibt es zwei prinzipielle Ansätze.

Interpreter/Interpretation. Das Programm (der AST) wird durchlaufen, und die Ausdrücke ausgewertet. Im Fall von Addition, werden die Operanden ausgewertet, dann die Addition ausgeführt usw. Ein Interpreter ist Teil der Source files.

Compiler/Compilation. Das Programm (der AST) wird in eine Maschinen-verarbeitbare Form gebracht welche direkt vom Prozessor ausgewertet werden kann. Die Compilation ist ein komplexer Prozess und wird deshalb in eine Reihe von Zwischenschritten (unter Zuhilfenahme von Zwischensprachen) ausgeführt. Z.B. wird der "high-level" AST in einen "low-level" AST umgewandelt, dann dieser in Zwischencode (z.B. nur noch jumps anstatt loops), dann wird der Zwischencode in Assembler umgesetzt usw.

Das Ziel heutzutage ist, dass ein Compiler weitesgehend Plattform-unabhängig ist. Deshalb wird anstatt Assembler/Maschinencode, Code für eine virtuelle Maschine erzeugt (siehe Java/JVM und C#/.Net).

Für unsere sehr einfache Programmiersprache (arithmetische Ausdrücke) betrachten wir eine sehr einfache virtuelle Maschine (VM).

Wir betrachten eine **stack-basierte VM** in welcher sämtliche Operationen mit Hilfe eines Stacks ausgeführt werden. Der Befehlssatz ist dadurch sehr (sehr) einfach und kann effizient umgesetzt werden. Als populäres Beispiel betrachte man WebAssembly.

Unsere stack-basierte VM verfügt über folgende Instruktionen (Befehle/Codes).

```
Push i

     schiebe (push) Konstante i auf den Stack

Plus

     hole Operanden von Stack
     fuehre Addition aus
     schiebe Ergebnis aud Stack

Mult

     hole Operanden von Stack
     fuehre Multiplikation aus
     schiebe Ergebnis aud Stack
```

Ein paar Beispiele, um zu zeigen, wie damit arithmetische Ausdrücke ausgewertet werden können.

Anstatt des ASTs, betrachten wir Ausdrücke in konkreter Syntax. Zuerst der Ausdruck in konkreter Syntax und dann die VM Instruktionen.

```
1 + 2

Push 1; Push 2; Plus;
```

Wir betrachten den Stack während der Abarbeitung der VM Instruktionen. Den Stack schreiben wir von links nach rechts. Links = top.

```
Initial:

Push 1; Push 2, Plus

Stack = []


1. Schritt

Push 2, Plus

Stack = [1]

2. Schritt

Plus

Stack = [2, 1]

3. Schritt


Stack = [3]
```

Ein komplexeres Beispiel.

```
1 * (2 + 3)

Push 1; Push 2; Push 3; Plus; Mult;
```

Die einzelnen Auswertungsschritte.

```
Initial:

Push 1; Push 2; Push 3; Plus; Mult;

Stack = []

1. Schritt
```

```
Push 2; Push 3; Plus; Mult;

Stack = [1]

2. Schritt

Push 3; Plus; Mult;

Stack = [2, 1]

3. Schritt

Plus; Mult;

Stack = [3, 2, 1]

4. Schritt

Mult;

Stack = [5, 1]

5. Schritt

Stack = [5]
```

Teilaufgabe SEMANTIK

Übersetzen Sie den AST in VM Instruktionen. Mehr Hilfestellungen gibt es in der Vorlesung.

Hinweis: Wie können Sie ihre Lösung testen? Vergleich des (finalen) Ergebnis Interpreter und Ausführung VM Instruktionen.

Source Files (Aufgabe 3)

Makefile ast.h, ast.cpp tokenizer.h, tokenizer.cpp parser.h, parser.cpp vm.h, vm.cpp testParser.cpp testVM.cpp

Makefile

```
CC=g++ --std=c++11
testParser : testParser.o parser.o tokenizer.o ast.o
    $(CC) testParser.o parser.o tokenizer.o ast.o -o testParser
testParser.o: testParser.cpp
    $(CC) -c testParser.cpp

testVM : testVM.o vm.o
    $(CC) testVM.o vm.o -o testVM
testVM.o: testVM.cpp
    $(CC) -c testVM.cpp


ast.o: ast.cpp ast.h
    $(CC) -c ast.cpp
tokenizer.o: tokenizer.cpp tokenizer.h
    $(CC) -c tokenizer.cpp
parser.o: parser.cpp parser.h ast.h tokenizer.h utility.h
    $(CC) -c parser.cpp
vm.o: vm.cpp vm.h
    $(CC) -c vm.cpp
```

NB. If you copy and paste the above Makefile, some tabs might not be copied. There need to be some tabs in the third, fith line and so on. Depending on your system, you may also need to set CC to your specific C++ compiler.

The following Makefile is kindly provied by one of the students and assumed to work under Linux with gcc and clang.

```
CXXFLAGS+= --std=c++11

build: testParser testVm

testParser : testParser.o parser.o tokenizer.o ast.o vm.o
    $(CXX) $(CXXFLAGS) testParser.o parser.o tokenizer.o ast.o vm.o -o
testParser

testParser.o: testParser.cpp

testVm : testVm.o vm.o testParser.o parser.o tokenizer.o ast.o
    $(CXX) $(CXXFLAGS) testVm.o vm.o parser.o tokenizer.o ast.o -o
testVM

testVm.o: testVm.cpp


ast.o: ast.cpp ast.h

tokenizer.o: tokenizer.cpp tokenizer.h

parser.o: parser.cpp parser.h ast.h tokenizer.h utility.h

vm.o: vm.cpp vm.h

clean:
    $(RM) *.o testParser testVm
```

utility

utility.h

```
// Utility stuff

#ifndef __UTILITY__
#define __UTILITY__

// Haskell's Maybe
template<typename T>
class Optional {
  bool b;
  T val;
public:
  Optional() : b(false) {}
  Optional(T v) : val(v), b(true) {}
```

```
  bool isJust() { return b; }
  bool isNothing() { return !b; }
  T fromJust() { return val; }
};

template<typename T>
Optional<T> nothing() {
  return Optional<T>();
}

template<typename T>
Optional<T> just(T v) {
  return Optional<T>(v);
}


#endif // __UTILITY__
```

ast

ast.h

```
// AST for exp


#ifndef __AST__
#define __AST__

#include <iostream>
#include <string>
#include <memory>

using namespace std;

class Exp {
public:
  virtual int eval() = 0;
  virtual string pretty() = 0;
};

class IntExp : public Exp {
  int val;
  public:
  IntExp(int _val) { val = _val; }
```

```
  int eval();
  string pretty();
};

class PlusExp : public Exp {
  std::shared_ptr<Exp> e1;
  std::shared_ptr<Exp> e2;
  public:
  PlusExp(std::shared_ptr<Exp> _e1, std::shared_ptr<Exp> _e2) {
      e1 = _e1; e2 = _e2;
  }
  int eval();
  string pretty();
};


class MultExp : public Exp {
  std::shared_ptr<Exp> e1;
  std::shared_ptr<Exp> e2;
  public:
  MultExp(std::shared_ptr <Exp> _e1, std::shared_ptr<Exp> _e2) {
      e1 = _e1; e2 = _e2;
  }
  int eval();
  string pretty();
};

// Short-hands

typedef std::shared_ptr<Exp> EXP;

EXP newInt(int i);

EXP newPlus(EXP l, EXP r);

EXP newMult(EXP l, EXP r);


#endif // __AST__
```

ast.cpp

```
#include <iostream>
#include <string>

using namespace std;
```

```
#include "ast.h"

int IntExp::eval() { return val;}

string IntExp::pretty() {
      return to_string(val);
}

int PlusExp::eval() { return e1->eval() + e2->eval(); }

string PlusExp::pretty() {
    string s("(");
    s.append(e1->pretty());
    s.append("+");
    s.append(e2->pretty());
    s.append(")");
    return s;
}


int MultExp::eval() { return e1->eval() * e2->eval(); }

string MultExp::pretty() {
    string s("(");
    s.append(e1->pretty());
    s.append("*");
    s.append(e2->pretty());
    s.append(")");
    return s;
  }

EXP newInt(int i) {
  return std::make_shared<IntExp>(i);
}

EXP newPlus(EXP l, EXP r) {
  return std::make_shared<PlusExp>(l,r);
}

EXP newMult(EXP l, EXP r) {
  return std::make_shared<MultExp>(l,r);
}
```

tokenizer

tokenizer.h

```
// Tokenizer for exp


#ifndef __TOKENIZER__
#define __TOKENIZER__

#include <iostream>
#include <string>
#include <vector>

using namespace std;

typedef enum {
  EOS,            // End of string
  ZERO,
  ONE,
  TWO,
  OPEN,
  CLOSE,
  PLUS,
  MULT
} Token_t;

string showTok(Token_t t);

// Elementary tokenize(r) class
class Tokenize {
  string s;
  int pos;
public:
  Tokenize(string s) {
         this->s = s;
     pos = 0;
  }

    // Scan throuh string, letter (symbol) by letter.
  Token_t next();
  vector<Token_t> scan();
  string show();

};


// Wrapper class, provide the (current) token.
class Tokenizer : Tokenize {
```

```
public:
  Token_t token;
  Tokenizer(string s) : Tokenize(s) { token = next(); }
  void nextToken() {
    token = next();
  }
};

#endif // __TOKENIZER__
```

tokenizer.cpp

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

#include "tokenizer.h"


string showTok(Token_t t) {
  switch(t) {
  case EOS:   return "EOS";
  case ZERO:  return "ZERO";
  case ONE:   return "ONE";
  case TWO:   return "TWO";
  case OPEN:  return "OPEN";
  case CLOSE: return "CLOSE";
  case PLUS:  return "PLUS";
  case MULT:  return "MULT";
  }
  // NOTE: The (clang) compiler is able to figure out that
  // along all control-flow paths, a return statement will be reached.
}

Token_t Tokenize::next() {
    if(s.length() <= pos)
      return EOS;

    while(1) {

       if(s.length() <= pos)
         return EOS;

       switch(s[pos]) {
```

```
        case '0': pos++;
                  return ZERO;
        case '1': pos++;
                  return ONE;
        case '2': pos++;
                  return TWO;
        case '(': pos++;
                  return OPEN;
        case ')': pos++;
                  return CLOSE;
        case '+': pos++;
                  return PLUS;
        case '*': pos++;
                  return MULT;
        default:  // we simply skip all other symbols !
                  pos++;
              break;
        }
    }
} // next


vector<Token_t> Tokenize::scan() {
    vector<Token_t> v;
    Token_t t;

    do {
      t = next();
      v.push_back(t);
    }
    while(t != EOS);

    return v;
} // scan


string Tokenize::show() {
    vector<Token_t> v = this->scan();
    string s;

    for(int i=0; i < v.size(); i++) {
      s += showTok(v[i]);
      if(i+1 < v.size())
    s += ";" ;         // Add delimiter
    }
    return s;
} // show
```

parser

parser.h

```
// (Top-down) Parser for exp

#ifndef __PARSER__
#define __PARSER__

#include "utility.h"
#include "ast.h"
#include "tokenizer.h"

class Parser {
  Tokenizer t;

  // E  ::= T E'
  Optional<EXP> parseE();

  // E' ::= + T E' |
  Optional<EXP> parseE2(EXP left);

  // T  ::= F T'
  Optional<EXP> parseT();

  // T' ::= * F T' |
  Optional<EXP> parseT2(EXP left);

  // F ::= N | (E)
  Optional<EXP> parseF();


public:
  Parser(string s) : t(Tokenizer(s)) { }

  Optional<EXP> parse() {
    Optional<EXP> e= parseE();

    return e;
  }
};
```

```
#endif // __PARSER__
```

parser.cpp

```
#include "utility.h"
#include "ast.h"
#include "tokenizer.h"
#include "parser.h"


// E  ::= T E'
Optional<EXP> Parser::parseE() {

    Optional<EXP> t = parseT();
    if(t.isNothing())
      return t;

    return parseE2(t.fromJust());
}


// E' ::= + T E' |
Optional<EXP> Parser::parseE2(EXP left) {

    if(t.token == PLUS) {
        t.nextToken();

    Optional<EXP> right = parseT();
    if(right.isNothing())
      return right;

    return parseE2(newPlus(left, right.fromJust()));
    }

    return just<EXP>(left);
}


// T  ::= F T'
Optional<EXP> Parser::parseT() {

    Optional<EXP> f = parseF();
    if(f.isNothing())
```

```
      return f;

    return parseT2(f.fromJust());
}


// T' ::= * F T' |
Optional<EXP> Parser::parseT2(EXP left) {
    if(t.token == MULT) {
      t.nextToken();

      Optional<EXP> right = parseF();
      if(right.isNothing())
    return right;

      return parseT2(newMult(left, right.fromJust()));
    }

    return just<EXP>(left);
}


// F ::= N | (E)
Optional<EXP> Parser::parseF() {
      switch(t.token) {
      case ZERO:
          t.nextToken();
          return just<EXP>(newInt(0));
      case ONE:
          t.nextToken();
          return just<EXP>(newInt(1));
      case TWO:
          t.nextToken();
          return just<EXP>(newInt(2));
      case OPEN: { // introduce new scope
          t.nextToken();
      Optional<EXP> e = parseE();
      if(e.isNothing())
        return e;
          if(t.token != CLOSE)
        return nothing<EXP>();
      t.nextToken();
      return e; }
      default: return nothing<EXP>();
      }

}
```

vm

vm.h

```
// Stack-based VM

#ifndef __VM__
#define __VM__

#include <vector>
#include <stack>

using namespace std;

#include "utility.h"


/*

Stack-based VM, instructions supported are:

  Push i
  Plus
  Mult

 */



typedef enum {
  PUSH,
  PLUS,
  MULT

} OpCode_t;


class Code {
public:
  OpCode_t kind;
  int val;

  // Nullary VM code (PLUS, MULT)
  Code(OpCode_t o) : kind(o) {}
  // Unary VM code (Push i)
```

```
  Code(OpCode_t o, int i) : kind(o), val(i) {}
};

// Short-hands

Code newPush(int i);

Code newPlus();

Code newMult();

class VM {
    vector<Code> code;
    stack<int> s;
  public:
    VM(vector<Code> c) : code(c) {}

    Optional<int> run();

};




#endif // __VM__
```

vm.cpp

```
#include "utility.h"
#include "vm.h"


Code newPush(int i) {
  return Code(PUSH, i);
}

Code newPlus() {
  return Code(PLUS);
}

Code newMult() {
  return Code(MULT);
}
```

```
Optional<int> VM::run() {

      // always start with an empty stack
      stack<int> d;
      s.swap(d);

      for(int i = 0; i < code.size(); i++) {
    switch(code[i].kind) {
    case PUSH:
      s.push(code[i].val);
        break;
    case MULT: {
      int right = s.top();
      s.pop();
      int left = s.top();
      s.pop();
      s.push(left * right);
      break;
    }
    case PLUS: {
      int right = s.top();
      s.pop();
      int left = s.top();
      s.pop();
      s.push(left + right);
      break;
    }
    }
      }

      if(s.empty())
    return nothing<int>();

      return just<int>(s.top());
} // run
```

testParser.cpp

```
#include <iostream>
#include <string>

using namespace std;


#include "parser.h"
```

```
#include "ast.h"



void display(Optional<EXP> e) {
  if(e.isNothing()) {
    cout << "nothing \n";
  } else {
    cout << (e.fromJust())->pretty() << "\n";
  }
  return;
}

void testParserGood() {

  /*
  display(Parser("1").parse());

  display(Parser("1 + 0 ").parse());

  display(Parser("1 + (0) ").parse());

  display(Parser("1 + 2 * 0 ").parse());

  display(Parser("1 * 2 + 0 ").parse());
  */

  display(Parser("(1 + 2) * 0 ").parse());

  display(Parser("(1 + 2) * 0 + 2").parse());
}

void testParser() {

  testParserGood();
}


int main() {

    testParser();

  return 1;
}
```

testVM.cpp

```cpp
#include <iostream>
#include <string>

using namespace std;

#include "vm.h"

void showVMRes(Optional<int> r) {
  if(r.isNothing())
    cout << "\nVM stack (top): empty";

  cout << "\nVM stack (top):" << r.fromJust();


}

void testVM() {

  {
    vector<Code> vc{
          newPush(1),
      newPush(2),
      newPush(3),
      newMult(),
      newPlus() };

    Optional<int> res = VM(vc).run();

    showVMRes(res);
  }

  {
    vector<Code> vc{
          newPush(2),
      newPush(3),
      newPush(5),
      newPlus(),
      newMult() };

    Optional<int> res = VM(vc).run();

    showVMRes(res);
  }
}


int main() {
```

```
  testVM();

  return 1;
}
```