Курс “C++ Basic”

Домашнее задание № 4 “Инженерный калькулятор”

If you lie to a compiler, it will get its revenge
Henry Spencer

Как задача “Hello, world” служит отличным примером для первого ознакомления с языком и инструментами сборки, так задача по разбору арифметического выражения из строки является отличным примером, на котором можно как закрепить различные темы связанные с языком программирования, так и познакомиться с базовыми идеями построения компиляторов.

В данном задании вам предстоит написать инженерный калькулятор, который принимает на вход текст с арифметическим выражением и в результате своей работы преобразует его в абстрактное синтаксическое дерево (АСД) (анг. AST - Abstract Syntax Tree). AST - это внутреннее представление программы внутри компилятора. В данной работе нужно будет распечатать полученную структуру на экран. В дальнейших работах мы будем развивать этот пример, снабжая его дополнительной функциональностью.

Схема компилятора

Классическая упрощенная схема компилятора представлена на рис. 1. Она включает в себя 2 основных блока - лексический и синтаксический анализаторы.

Рис 1. Упрощенная схема компилятора

На вход лексическому анализатору (лексеру) подаются символы, на выходе он формирует токены, которые представляют собой элементарные кирпичики. Например, число 123 состоит из 3 символов, но представляют собой один токен типа целое число.

Алгоритм работы лексического анализатора

Лексер построен на основе логики работы конечного автомата. Его код описан в заголовочном файле lexer.h и файле реализации lexer.cpp. Основная логика находится внутри метода `Lexer::next_token()`. Этот метод должен разобрать следующий токен в потоке входных символов и вернуть его тип. Для кодирования типа токена используется типизированное перечисление (`enum class`) Token. Подробней при enum

class можно почитать в статье [4]. В отличии от обычного перечисления в стиле C (`enum`), типизированное перечисление позволяет компилятору контролировать типы при присваивании и сравнении, а также следить за покрытием всех возможных переходов в `switch` конструкциях.

Также лексер хранит свое внутреннее состояние, закодированное с помощью типизированного перечисления `State`. Для разбора некоторых токенов, например чисел и имен, лексеру необходимо считать из входного потока несколько символов. Так для получения значения числа из символьного представления необходимо читая цифры слева направо постоянно накапливать результат их в аккумуляторе умножая текущий результат на 10 (сдвиг влево) и прибавляя новую цифру.

Схема переходов между состояниями лексера изображена на рисунке 2. Каждый новый символ, прочитанный из входного потока, может перевести лексер в новое состояние или оставить в предыдущем (переход по типу петли). Empty - начальное состояние, End - заключительное. Подписи на стрелках обозначают набор символов, по которому осуществляется переход.

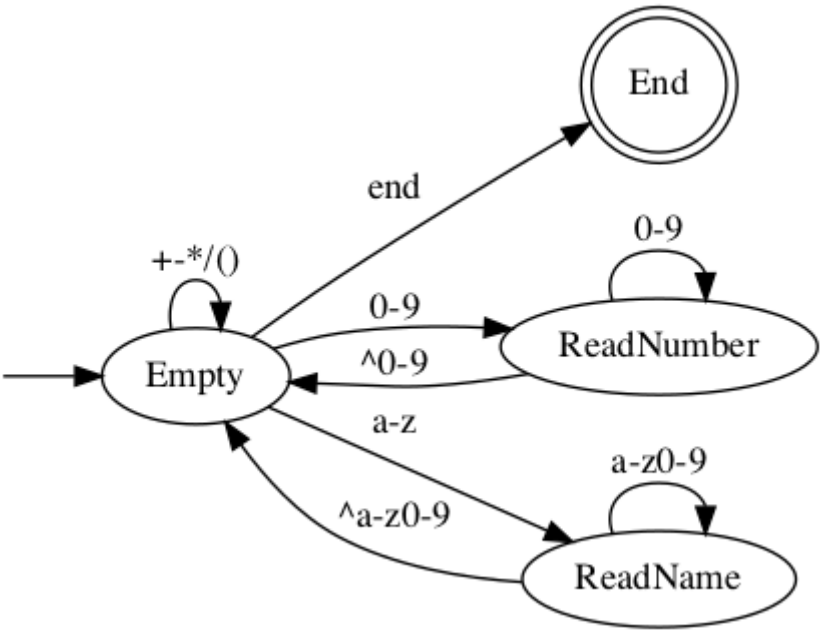

Рис 2. Конечный автомат переходов состояний лексера

Разберем следующий пример. На вход лексера подается строка 12 + a. Изначальное состояние Empty. Считываем первый символ ‘1’. Из состояния Empty имеется переход в состояние ReadNumber подписанный как 0-9 (что означает любой символ от ‘0’ до ‘9’ или любая цифра). Состояние меняется на ReadNumber, а переменной члену класса number_ присваивается значение 1. По следующему символу ‘2’ остаемся в том же состоянии ReadNumber, а переменной number_ присваивается значение 10 * 1 + 2 = 12. Следующий символ ‘+’ переводит автомат в состояние Empty (переход по стрелке ^0-9, что читается как любые символы кроме диапазона от ‘0’ до ‘9’). При этом метод next_token() возвращает значение Token::Number, сигнализируя, что из входного потока было прочитано число. Считанный символ ‘+’ запоминается, чтобы использовать его при следующем вызове метода.

При следующем вызове next_token() мы обрабатываем сохраненный символ ‘+’ и остаемся в текущем состоянии Empty, возвращая токен Operator. Последующий вызов приведет нас в состояние ReadName, поскольку на встретился символ ‘a’. А из Empty есть переход в ReadName по диапазону символов ‘a-z’. После появления в потоке признака конца файла или конца строки лексер переходит в заключительное состояние End и всегда возвращает специальный токен End, сигнализируя, что данные во входном потоке закончились.

Алгоритм работы синтаксического анализатора

Синтаксический анализатор получает на вход поток токенов и преобразуется его в АСД как это показано на примере на рисунке 3.

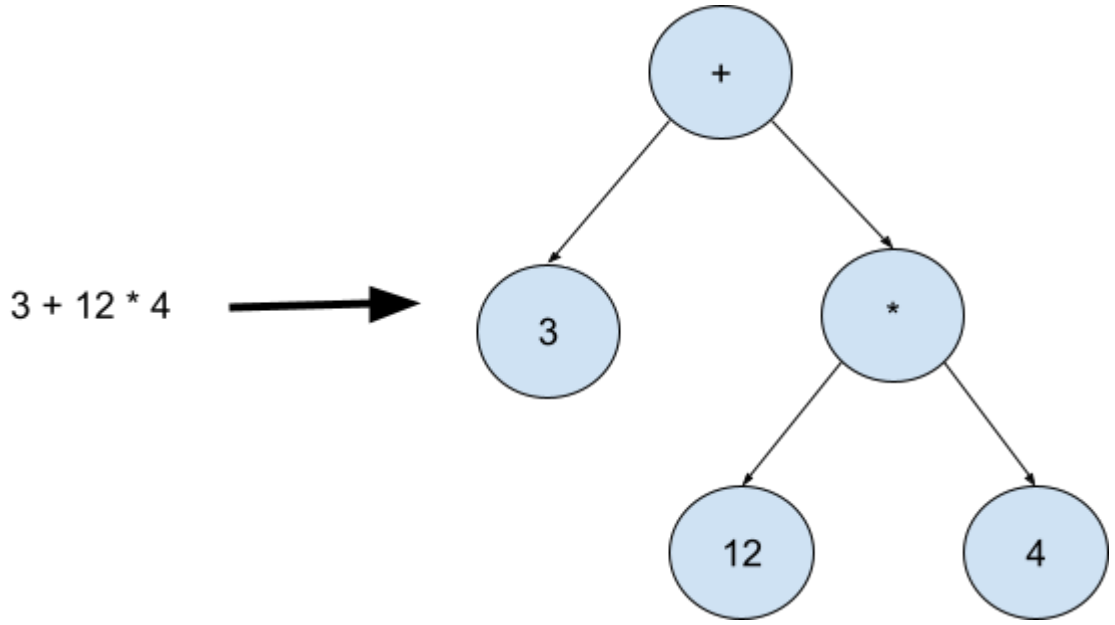

Рис 3. Пример арифметического выражения и соответствующего ему АСД

Его реализация описана в файлах parser.h и parser.cpp. Реализация метод Parser::parse() использует вспомогательные защищенные методы expr(), term() и prim(). Эти методы реализуют логику работы алгоритма рекурсивного спуска для разбора арифметических выражений. Рассмотрим следующую нотацию, с помощью которой можно описать произвольное абстрактное арифметическое выражение. Пусть выражение E (expression) может состоять из:

1. Одного слагаемого T (term)
2. Слагаемого T + другое выражение E
3. Слагаемого T - другое выражение E.

Эти 3 варианта можно записать в виде E -> T | T + E | T - E. Здесь символ ‘|’ обозначает ‘или’. Такое рекурсивное определение для выражения E позволяет описать выражение с любым числом слагаемых.
Само слагаемое может состоять из

1. Базового элемента (число или имя переменной) P (prim)
2. Базового элемента * T
3. Базового элемента / T

или в более короткой нотации T -> P | P * T | P / T.
Для базового элемента можно описать P -> Number | Name.
Например, выражение 3 * 4 / 5 можно получить из T с помощью серии подстановок описанным выше правилам:
T -> P * T -> P * P / T -> P * P / P

Все вместе описывает грамматику языка арифметических выражений (без скобок) в форме Бэкуса-Наура (БНФ)
E -> T | T + E | T - E
T -> P | P * T| P / T
P -> Number | Name

Рассмотрим на примере выражения 3 + 4 - 5, как на основе полученной БНФ можно построить парсер. Начинаем мы с вызова метода для разбора выражения E (expr). Считываем первое слагаемое term(), которое создает узел типа Number и возвращает в виде указателя на ASTNode. Эволюция AST при разборе данного примера показана на рисунке 4. Далее считываем оператор сложения ‘+’ и следующее слагаемое ‘4’. Формируем из них дерево, в корне которого узел Add, а в его поддеревьях узлы Number(3) и Number(4). Следующая операция Sub становится новым корнем дерева с узлом Number(5) в правом поддереве. Поскольку токены во

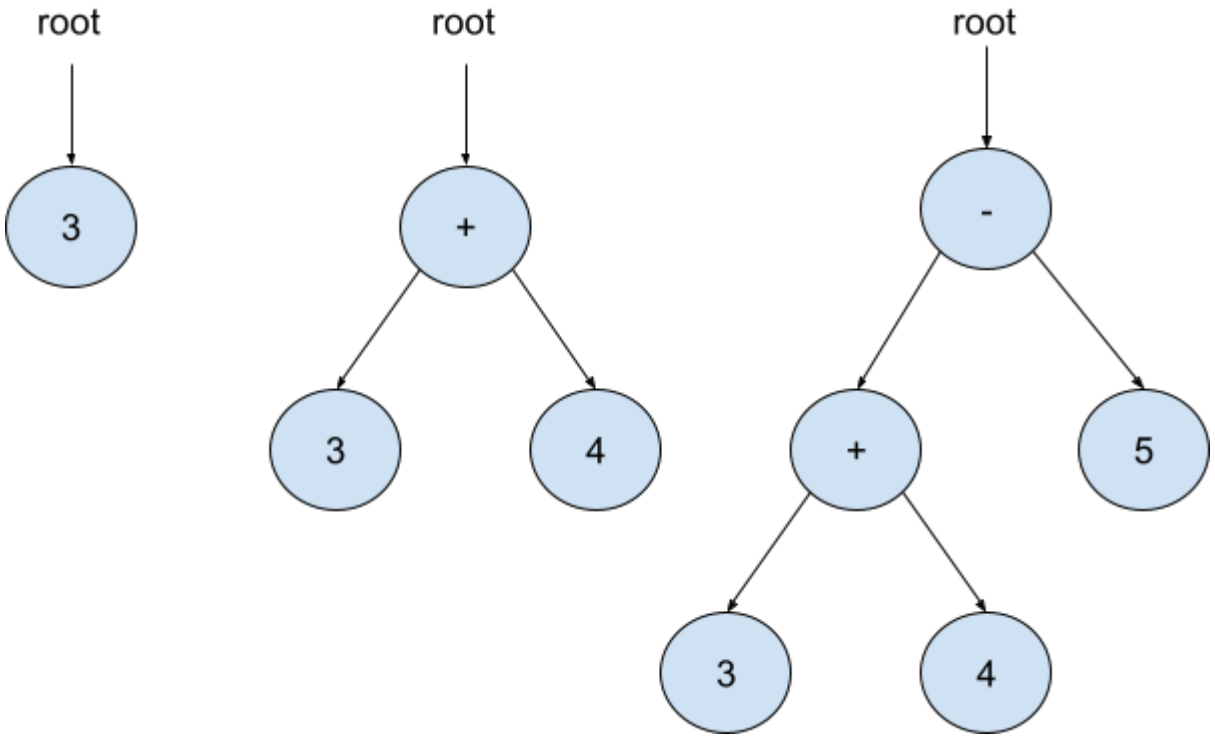

Рис 4. Пример формирования АСД.

входном потоке закончились метод expr() возвращает указатель на текущий корень АСД.

Подробнее с построением лексера и парсера для арифметических выражений можно познакомиться в книге [1].

Задание

Исходные коды лексера и пример его использования следует взять из архива, приложенного к материалам занятия. Также реализован парсер на основе метода рекурсивного спуска. Для полноты реализации парсера не хватает объявления наследников ASTNode, которые нужно будет реализовать. После добавления нужных классов можно подключить компиляцию парсера в CMakeLists.txt (раскомментировать строку) и раскомментировать использование в main.cpp

Вам нужно написать программу, которая конструирует АСД и распечатывает его в текстовом виде.

В качестве базового класса для узла дерева следует использовать интерфейс из листинга 1. Файл с объявлением и реализацией имеется в приложенном архиве.

```
class ASTNode {
public:
   // Конструктор листа (узел без дочерних узлов)
   explicit ASTNode(const std::string &repr);

   // Конструктор узла (с одним или двумя дочерними узлами)
   ASTNode(const std::string &repr,
           ASTNode *lhs,
           ASTNode *rhs);

   // Представления имени узла в виде строки
   std::string repr() const;

   // Печать всего дерева в текстовом виде
   void print(std::ostream &out) const;
};
```

Листинг 1. Интерфейс класса ASTNode из astnode.hpp

Нужно объявить наследников класса ASTNode, которые будут представлять следующие типы узлов:

- **Number** - целочисленная константа (нет дочерних узлов) - пример реализации в архиве в файле number.hpp
- **Add, Sub, Mul, Div** - арифметические операции (два дочерних узла)
- **Variable** - имя переменной (нет дочерних узлов)

Для печати АСД можно использовать простой рекурсивный алгоритм, который выведет дерево уложенное набок. Например, выражение 3 + 12 * 4 распечатается в следующем виде. Пример реализации алгоритма в файле astnode.cpp

```
        3
+
                12
        *
                4
```

Критерии выполнения

Для зачета необходимо выполнить обязательные критерии. Дополнительные критерии можно не выполнять, но они дадут больше возможностей для развития навыком программирования на C++.

Обязательная часть

- Предоставлен исходный код решения в виде архива или ссылки на репозиторий
- В репозитории есть CMakeLists.txt для сборки проекта
- Проект собирается с помощью cmake без ошибок и предупреждений со следующими опциями
    - Для gcc, clang: -Wall -Wextra -Werror -pedantic
    - Для Visual Studio: /W4 /Wx
- В результате сборки получается один исполняемый файл
- При реализации классов учтено правило 3 (рассматривали на вебинаре)
- Программа перед завершением высвобождаем всю динамическую память
- При запуске программа считывает одно выражение и завершается
- Правильно выполняются следующие примеры

Input:3
Output:
3

Input: 5 + 12
Output:

```
        5
+
        12
```

Input: 3 + 12 * 4
Output

```
        3
```

```
+
        12
    *
        4
```

Input: 2 * 3 - 6 / 2
Output:

```
        2
    *
        3
-
        6
    /
        2
```

Input: a*a + 2*a*b + b*b
Output:

```
            a
        *
            a
    +
                2
            *
                a
        *
            b
+
        b
    *
        b
```

Дополнительная часть

- структурируйте программу с помощью разнесения логики по разным файлам с исходными кодами. Разделите объявления и реализации классов. Объявления поместите в заголовочные файлы, а реализацию в cpp.
- добавьте поддержку скобочных выражений. Для реализации такой логики можно рассмотреть алгоритм построения парсера на основе метода рекурсивного спуска. Например 3 * (2 + 4) должно вывестись как

```
3
*
        2
    +
        4
```

- Добавьте в лексер и парсер обработку ошибок. Например, следующие выражения должны порождать сообщение об ошибке и завершаться с соответствующим кодом возврата:

- 12abc
- * 3
- 12(+2)
- a b
- 12 43
- 3 +

Дополнительные материалы

1. **Б. Страуструп**. Язык программирования C++. Пример калькулятор
2. **А. Ахо, Р. Сети, Дж. Ульман**. Компиляторы, принципы, технологии и инструменты (aka книга дракона :)
3. Kaleidoscope: Implementing a language with LLVM
4. Типы struct, union и enum в modern C++