ENUMERATE

Минимум о перечисляемых типах и объединениях в языке C

К. Владимиров, Intel, 2019
mail-to: konstantin.vladimirov@gmail.com

Константы времени исполнения

- Модификатор const означает, что помеченная им переменная не может измениться во время выполнения программы

```
const int i = 5;
i = 2; // ошибка
```

- Но при этом её значение может быть получено очень поздно

```
int a[20];
fill(a);
const int j = a[0];
```

- Поэтому считается, что их значение неизвестно на этапе компиляции
- Это имеет одно печальное последствие

Константы времени исполнения

- Такие константы не могут быть использованы для размера массивов

```
const int asize = 5;

int a[asize]; // ошибка
```

- Дело в том, что размер массива должен быть известен на этапе компиляции
- Если не рассматривать препроцессор, язык C имеет одну возможность сделать константу времени компиляции: перечислимый тип

```
enum { asize = 5 };

int a[asize]; // всё хорошо
```

- Настало время поговорить о них подробнее

Перечислимый тип

- Главное предназначение enumerations: задавать синонимы для перечислений

```
enum weekday { monday, tuesday, wednesday, thursday,
               friday, saturday, sunday };

enum state { Working, Failed, Freezed };

enum year { Jan, Feb, Mar, Apr, May, Jun, Jul,
            Aug, Sep, Oct, Nov, Dec };
```

- Перечисление определяет тип, который можно использовать в программе

```
enum weekday w = monday;

assert (w < sunday);
```

Перечислимый тип

- Все значения перечислимых типов – целые числа (как int)
- По умолчанию они начинаются с нуля

```
enum state { Working, Failed, Freezed };

assert(Working == 0);
enum state w = Failed;
assert(w == 1);
```

- Но можно указывать любые значения

```
enum state { Working = 6, Failed = 12, Freezed };

assert(Freezed == 13); // не указанные значения растут как +1
```

Неименованные перечислимые типы

- Перечисляемый тип не задаёт область видимости, поэтому нельзя дублировать перечислители

```
enum result { Ok = 0, Failed = -1};
enum state { Working = 0, Failed, Freezed }; // ошибка
```

- До некоторой степени каждый перечислимый тип это просто набор констант. Поэтому если не нужны объекты этого типа, его можно никак не называть

```
enum { ASIZE = 10, BSIZE = 20 };

int arr[ASIZE];
int s = ASIZE - 1; // нормально, перечисление неименованное
```

Именованные перечислимые типы и int

- Перевод из именованных перечислимых типов в int и обратно считается дурным тоном

```
enum state { Working = 6, Failed = 12, Freezed };

int t = 4;
enum state s = t; // можно, но сомнительно
```

- Операции над именованными перечислимыми типами – тоже дурной тон

```
enum state s = Working;

s += 1; // можно, но ещё более сомнительно
```

- Rule of thumb: именованные перечисления лучше всего только сравнивать на равенство и неравенство и оперировать только внутри их значений

Перечислимые типы и функции

- Именованные перечислимые типы могут быть как аргументами, так и результатами функций

```
enum State { STATE_OK = 0, STATE_FAILED = -1 };

enum State get_state();

void set_state(enum State s);
```

- Часто, это лучше, чем возвращать int, поскольку естественным образом документирует возвращаемые значения
- Разумеется, и передавать и возвращать можно и по указателю

```
void modify_state(enum State *s);
```

Объединения

- Одним из главных применений перечислимых типов являются объединения
- Объединение очень похоже на структуру, только все поля в нём лежат по одному адресу

```
struct S {
  int x;
  char c;
};

union U {
  int x;
  char c;
};
```

Объединения

- Одним из главных применений перечислимых типов являются объединения
- Разумеется, брать из объединения можно только то, что туда положил

```
union IntChar {
  int x;
  char c;
};

union IntChar ic;

ic.c = 'a';
int y = ic.x; // крайне дурной тон (особенно в C++)
```

- И здесь удобно использовать перечисления

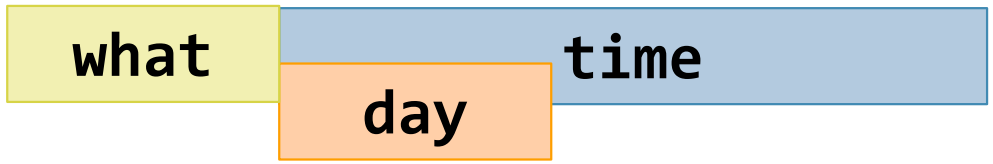
Объединения

- Здесь показана структура с объединением и перечислением

```
enum DTS { DT_DAY = 0, DT_TIME };

struct DT {
  enum DTS what;
  union DayOrTime {
    int day;
    time_t time;
  } u;
}

struct DT d1 = { .what = DT_DAY, .u.day = 42 }; // C style
struct DT d2 = { DT_DAY, 42 }; // C++ compatible
```



Пример: лексический анализ

- Нужно ввести скобочное выражение, содержащее четыре основных операции

```
(2 + 3) * 5 - 7
```

- Казалось бы это несложно, но есть проблемы
- Пробелы могут быть введены лишние или не введены вообще

```
(2+     3)*56 - 7
```

- Нужна диагностика для ошибок

```
(2 + 3] * 56 - Z
```

- Как представить считанное выражение?

Идея: массив лексем

- Лексема это операция, скобка или число

```
enum lexem_kind_t { OP, BRACE, NUM };
enum operation_t { ADD, SUB, MUL, DIV };
enum braces_t { LBRAC, RBRAC };

struct lexem_t {
  enum lexem_kind_t kind;
  union {
    enum operation_t op
    enum braces_t b;
    int num;
  } lex;
};
```

"(2+ 3)*56 – 7"

(	2	+	3	)	*	56	-	7

Problem LX

- Напишите функцию, которая делает из строки массив лексем

```
struct lex_array_t {
  struct lexem_t *lexems;
  int size, capacity;
};

struct lex_array_t lex_string(const char *str) {
  // TODO: your code here
}
```

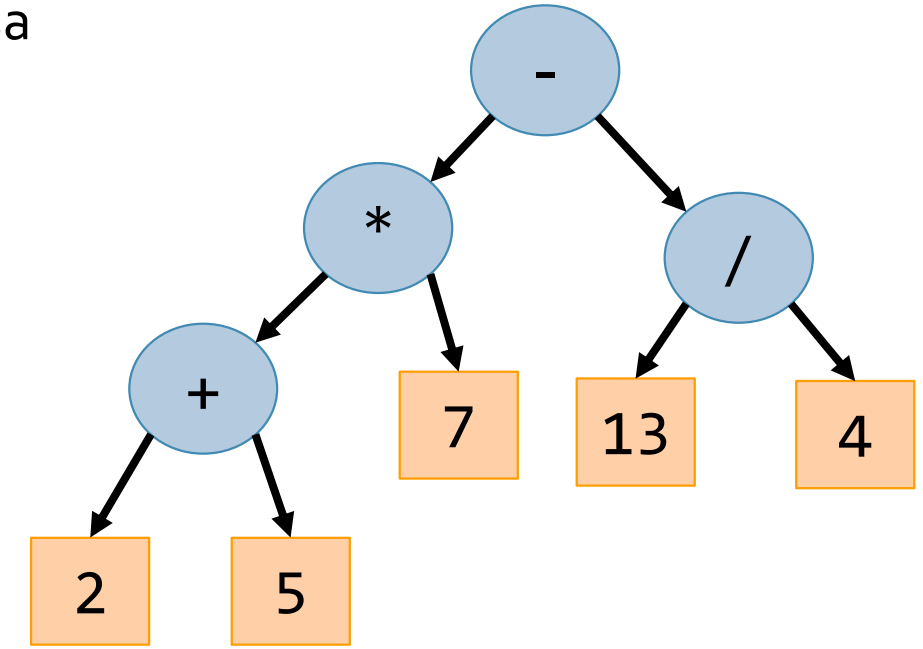
- Пробелы и пробельные символы ничего не значат
- При ошибке разбора должно печататься сообщение и завершаться программа

Пример: синтаксические деревья

- Выражению `((2 + 5) * 7 - 13 / 4)` соответствует дерево
- Удобный тип для содержимого узла такого дерева

```
enum node_kind_t { NODE_OP, NODE_VAL};

struct node_data_t {
  enum node_kind_t k;
  union {
    enum operation_t op;
    int d;
  } u;
};
```



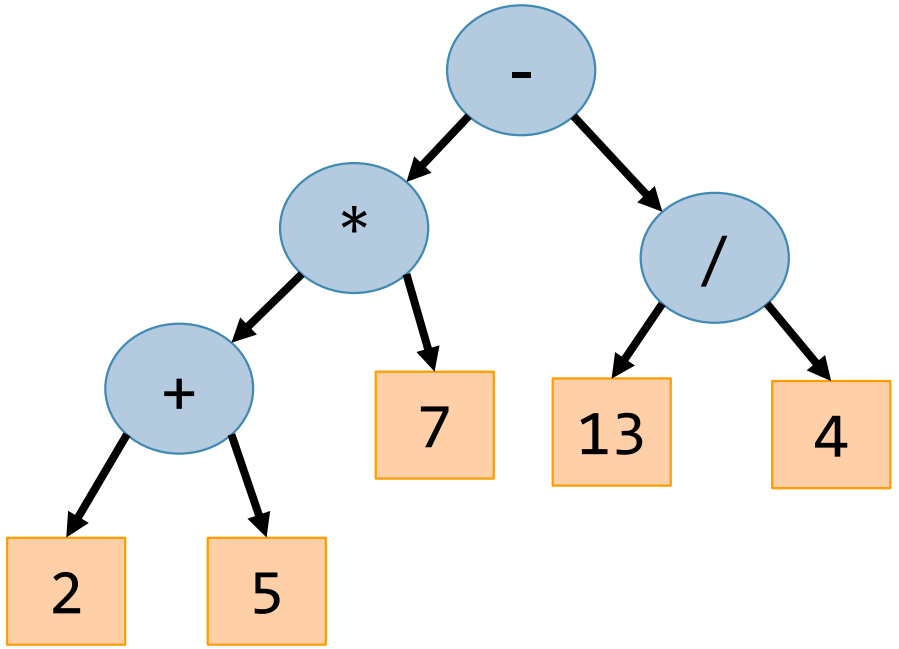
- Очевидно ли вам как, имея построенное дерево, вычислить выражение?

Пример: синтаксические деревья

- Выражению `((2 + 5) * 7 - 13 / 4)` соответствует дерево
- Удобный тип для узла такого дерева

```
struct node_t {
  struct node_t *left, *right;
  struct node_data_t data;
};
```

- Вычисление это просто postorder обход:
- сначала вычисляется левое поддерево
- потом правое
- потом применяется сама операция



Обсуждение

- Как перейти от массива лексем к синтаксическому дереву?

Синтаксический разбор

- Процесс построения синтаксического дерева из лексем называется синтаксическим разбором (parsing)
- Он зависит от грамматики. Например для простейших выражений

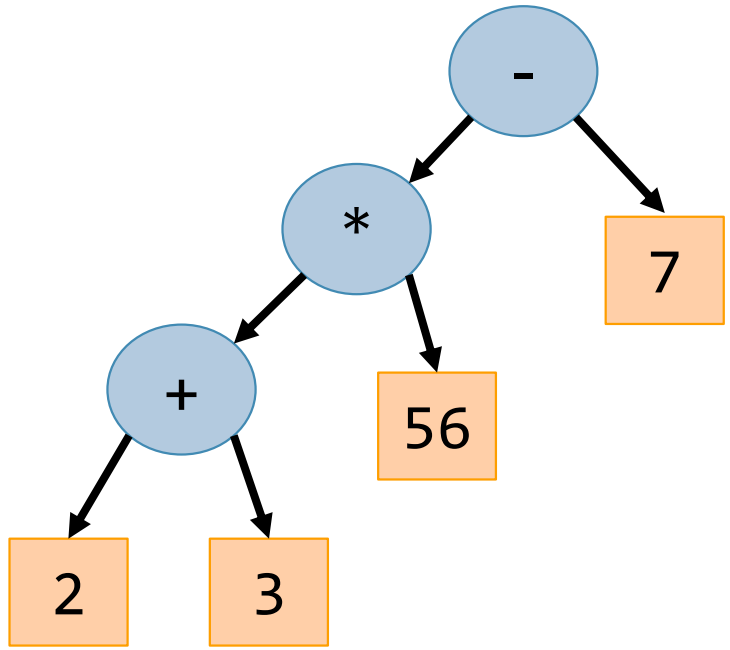
```
expr ::= mult {+, -} expr | mult
mult ::= term {*, /} mult | term
term ::= ( expr ) | number
```

- Здесь терминальные символы operation, number и скобки соответствуют типам лексем в массиве
- Каждый нетерминал (курсивом) соответствует рекурсивному вызову функции

Пример разбора

(	2	+	3	)	*	56	-	7

```
expr
mult – expr
term * term – mult
( expr ) * 56 – term
( mult + expr ) * 56 – 7
( term + mult ) * 56 – 7
( 2 + term ) * 56 – 7
( 2 + 3 ) * 56 - 7
```



Problem ST – синтаксическое дерево

- На вход приходят выражения, содержащие числа и арифметические операции

```
2 + 2 * (3 – 4) / 6
```

- Все операции имеют обычный приоритет и вычисляются слева направо

```
2 + 2 * 3 → 8
(2 + 2) * 3 / 5 * 2 → 12 / 5 * 2 → 2 * 2 → 4
63 + 86 / 3 / 5 * ( 13 - 69 ) → 53 + 5 * ( 13 - 69 ) → -217
```

- Необходимо выдать на выход значение для каждого из выражений
- Используйте лексический анализ из проблемы LX