1 Абстрактные синтаксические деревья

1.1 Управление памятью на основе регионов

1.1.1 Мотивировка

Текущая реализация абстрактного синтаксического дерева имеет следующие недостатки:

1. Выделение памяти стандартным методом может значительно фрагментировать оперативную память, затрудняя доступ к ней.
2. Любое выделение и удаление памяти требует вмешательства системных вызовов, что может стать причиной дополнительных издержек во время работы программы.
3. Программист не имеет возможности ручного управления выделяемой им памятью.

Избавиться от этих недостатков можно используя различные оптимизации. В рамках этой работы воспользуемся управлением памятью на основе, так называемых, регионов (арен, зон) [1].

Под регионом далее будем понимать непрерывную область памяти, содержащую внутри себя объекты. При запуске программы выделим регион некоторого размера, при необходимости увеличивая его размер в некоторое постоянное число раз.

Этот подход имеет следующие преимущества:

1. Элементы располагаются последовательно, в связи с чем минимизируется фрагментация и упрощается доступ к объектам.
2. Выделение и освобождение памяти выполняется с минимальными издержками.
3. Программисту предоставляется большая свобода для управления выделенной памятью.

1.1.2 Построение

Формально определим требования к системе:

1. Регион должен представлять из себя некоторый непрерывный участок размера n байт (в начальный момент времени размер равен некоторой начальной величине n_0).
2. При обращении к региону он должен предоставить k байт памяти и вернуть некоторый идентификатор этого участка для последующего обращения.

3. При заполнении региона должна быть возможность увеличить объем доступной памяти в некоторое число раз, которое далее будем называть коэффициентом увеличения.
4. Должна быть доступна возможность эффективного освобождения всей выделенной регионом памяти.

Единственной сложной операцией над регионом является его увеличение. Так как выделение нового участка потенциально может сопровождаться изменением адресов объектов, то необходимо организовать доступ к ним независимо от первоначального адреса. Для этого для каждого объекта будем получать доступ к нему через некоторый индекс.

Кроме того, коэффициент увеличения должен быть выбран таким образом, чтобы был соблюден баланс между оптимальным объемом выделенной памяти и частотой системных вызовов.

1.1.3 Определение структуры

Определим нашу структуру следующим образом:

```
typedef struct arena {
    // Указатель на начало региона
    struct node* arena;
    // Размер региона
    unsigned int size;
    // Объем выделенной регионом памяти
    unsigned int allocated;
    } arena;
```

1.1.4 Инициализация

Теперь определим функцию `arena_construct`, выполняющую начальную инициализацию состояния региона:

```
int arena_construct (arena* arena) {
    // Начальный размер региона равен некоторой постоянной, равной
   DEFAULT_ARENA_SIZE
    arena->size = DEFAULT_ARENA_SIZE;
    arena->allocated = 0;
    // Выделим необходимое число памяти
    arena->arena = malloc(sizeof(node) * DEFAULT_ARENA_SIZE);
    // Если выделение прошло неудачно - вернем в качестве кода ошибки отличное
    ↪ от 0 значение.
```

```
    if (arena->arena == NULL) {
        return (!0);
    }
    return 0;
}
```

1.1.5 Выделение памяти

После выделения некоторого объема памяти возможно обращение к ней. Определим это обращение с помощью функции `arena_allocate`:

```
int arena_allocate (arena* arena, unsigned int count) {
    // Если места в регионе недостаточно
    if (arena->allocated + count >= arena->size) {
        // Определим новый размер региона
        unsigned int newSize = MULTIPLY_FACTOR * arena->size;
        // Выделим регион большего размера и освободим ранее занятую память
    ↪ node*
        newArena = realloc(arena->arena, newSize * sizeof(node));
        if (NULL == newArena) {
            return -1;
        }
        arena->arena = newArena;
        arena->size = newSize;
        }
    // В качестве результата вернем индекс первого свободного участка региона
    unsigned int result = arena->allocated;
    // Сместим индекс на объем выделенной памяти
    arena->allocated += count;
    // Вернем результат
    return result;
}
```

Отметим, что наиболее часто значением `MULTIPLY_FACTOR` оказывается числа 1.5 и 2. Это позволяет достичь амортизационно константного времени выполнения операции выделения памяти [2].

1.1.6 Освобождение выделенной памяти

Наконец, реализуем освобождение выделенной региону памяти с помощью функции `arena_free`

```
void arena_free (arena* arena) { if (arena->arena != NULL) free(arena->arena);
    arena->arena = NULL; }
```

1.1.7 Модификация абстрактного синтаксического дерева

Осталось изменить исходный код программы, чтобы обеспечить выделение памяти с помощью полученной нами структуры данных.

Для этого воспользуемся директивой `%param` и заявим в качестве параметра переменную типа `arena*`. В функциях `eval`, `newnum`, `newast` внесем изменения, чтобы обеспечить выделение памятью с помощью написанных ранее функций.

С полным кодом программы можно ознакомиться в приложении А.

1.1.8 Сборка проекта

Теперь проект можно собрать, незначительно изменив `Makefile`:

```
1 calc.out: calc.l calc.y arena_ast.h bison -d calc.y flex calc.l cc -o $@
2     calc.tab.c lex.yy.c arena_ast.c arena.
```

и запустить. Результат работы программы представлен на рис. 6

пикча

2 Сравнение полученных реализаций

Проведем анализ производительности полученных версий анализатора. В качестве данных для тестирования возьмем выражения вида $2 + 2 + 2 \cdot \cdot \cdot + 2$ для $n = 1 \ldots 100$ с шагом 1. Для вычисления времени выполнения воспользуемся библиотекой `time` Python 3.9.5. Автоматизацию обеспечим с помощью библиотеки `subprocess`. Получим следующий код:

ЗАКЛЮЧЕНИЕ

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 *Wang, D. C.-A.* Managing memory with types / D. C.-A. Wang. — Princeton University, 2002. — Pp. 29–55.

2 Facebook open-source library documentation [Электронный ресурс]. — URL: `https://github.com/facebook/folly/blob/main/folly/docs/FBVector.md` (Дата обращения 25.05.2021). Загл. с экр. Яз. англ.