Разработка объектно-ориентированного языка со сборкой мусора

Студент: Масягин М.М.
Научный руководитель: Синявин А.В.

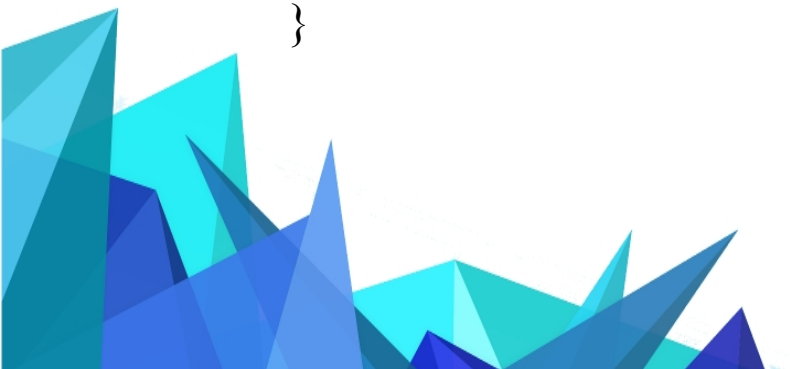
Постановка задачи

- Разработать спецификацию собственного объектно-ориентированного языка;
- Написать для спроектированного языка интерпретатор;
- Создать и протестировать подсистему сборки мусора;

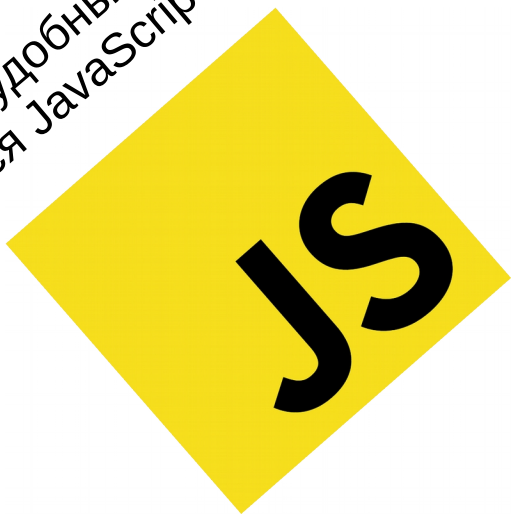
Дизайн языка

- С-подобный синтаксис;
- Объекты-мапы аля JavaScript;
- Автоматическая сборка мусора;
- Поддержание ссылок всегда актуальными;
- Сильная динамическая неявная типизация;
- Максимальная простота и однозначность языка:
  - отсутсвие побочных результатов операций;
  - объявление не более одной переменной за раз;
  - отсутствие goto-меток, операторов ++, --, null;

```
function test() {
	let a = {
		b : [1, 2, 3, 4, 5, 6, 7, 8, 9, -1]
		}; // Инициализация объекта и вложенного в него массива.
	a.b[9] = 10;
	a.c = a; // Циклическая зависимость.
	let i = -1;
	let sum = 0;
	while (i < 1000) { // while.
		i = i + 1;
		if (i == 10) { // if-then-else.
			break; // break.
		} else if (a.b[i] % 2 != 0) {
			continue; // continue.
		} else {
			sum = sum + a.c.b[i] * a.c.b[i];
		}
	}
	return sum;
}
```

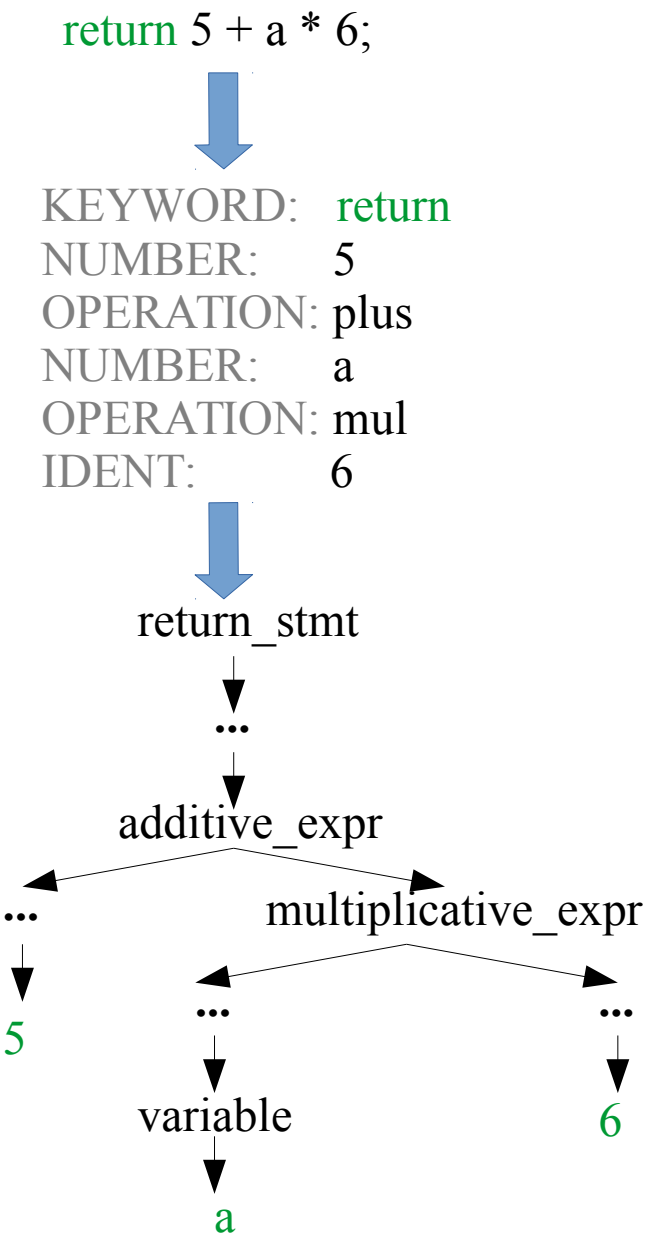


Я хотел создать удобный и современный ЯП, а получился JavaScript...



Front-end

- Собственноручно написанные лексер и парсер;
- Лексер способен работать “параллельно” с парсером и восстанавливаться при невалидных лексемах (игнорировать их);
- Парсер использует предсказывающий метод рекурсивного спуска (без возвратов) и при ошибках выводит рекомендации по их исправлению; Время работы - линейное;



Генератор байт-кода

- Поиск неинициализированных переменных;
- Поиск *break* и *continue* вне циклов;
- Использование пула констант;
- Оптимизации;

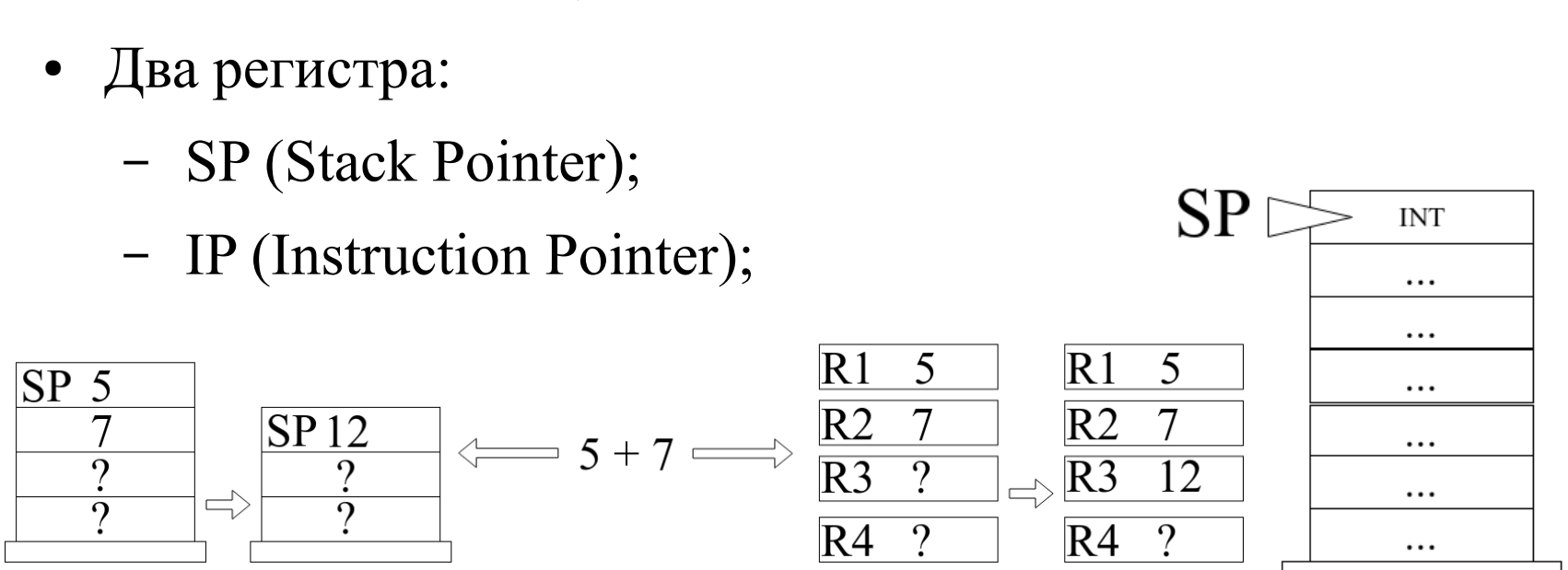
```
return a.b[i][1];
```

```
CONSTANT POOL:
INT_CNST IDX = 1 VALUE = 1
FIELDREF_CNST IDX = 1 VALUE = b

OP_CODES:
GET_LOCAL 1
CONSTANT 1
GET_HEAP 0 3 1 2 1 1 1 0
RETURN
```

Виртуальная машина

- Стековая архитектура ВМ;
- Два регистра:
  - SP (Stack Pointer);
  - IP (Instruction Pointer);

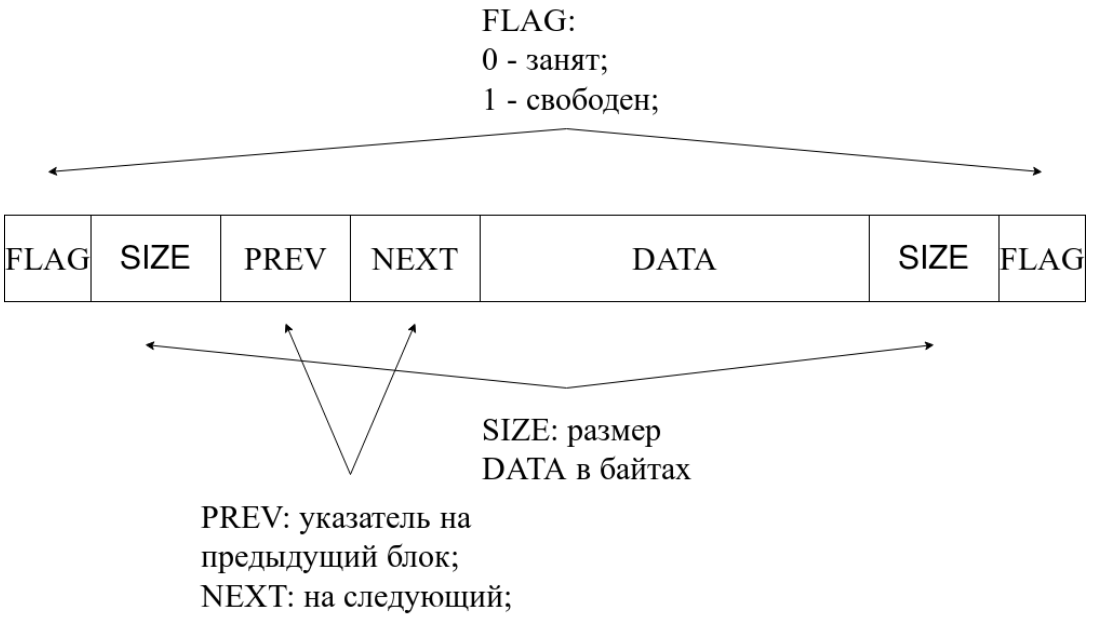


POP 5
POP 7
ADD 5 7 res
PUSH res

ADD R1, R2, R3

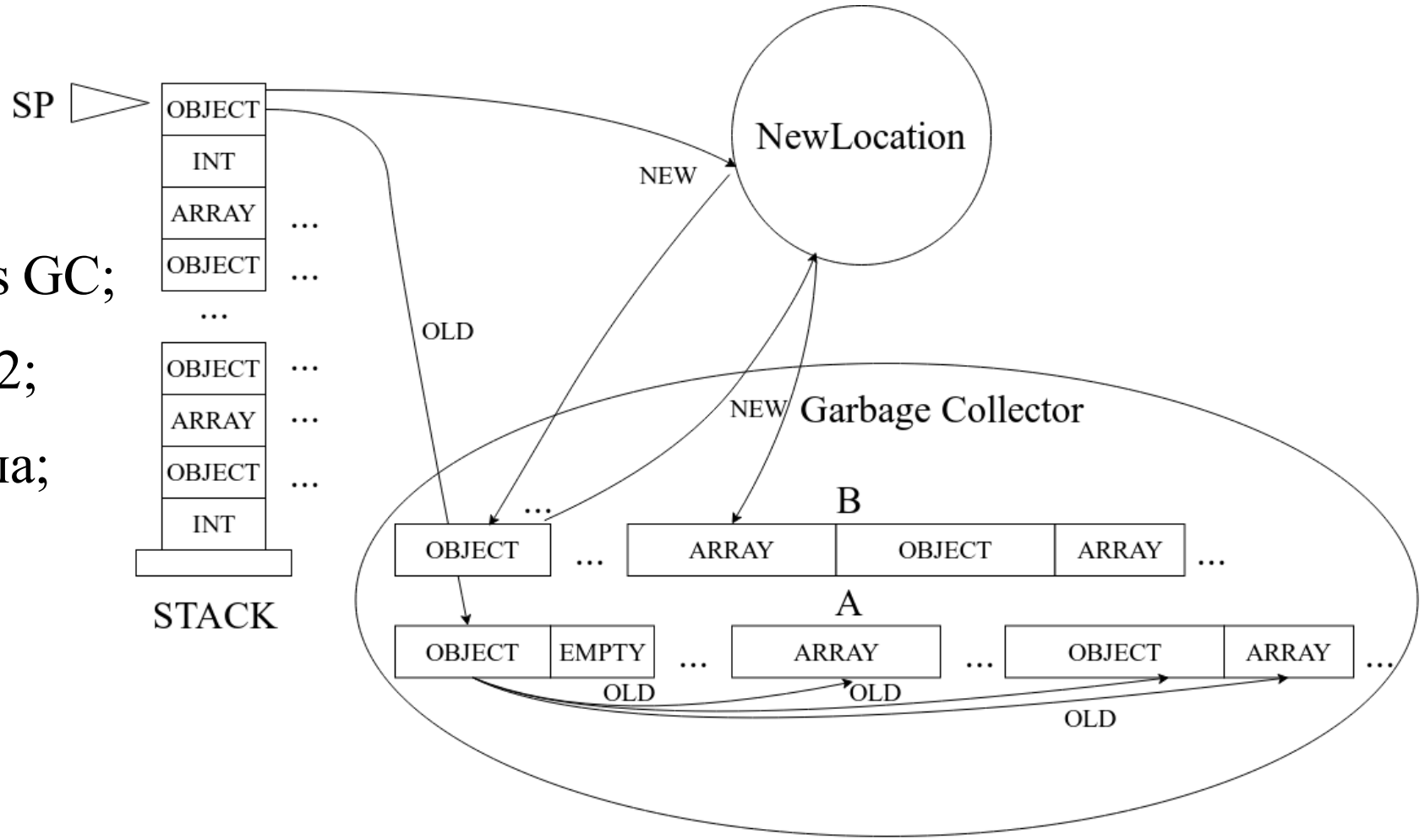
Аллокатор

- Аллокатор – двусвязный список на массиве;
- Списки *free* и *busy*;
- Методы *malloc*, *realloc*, *free*;
- Стратегия “наилучшего подходящего”;
- Сборка мусора при *realloc*;



Сборщик мусора

- Cheney’s GC;
- realloc x2;
- Байт типа;



Реализация

- Pure C99 (практически C89);
- Нет зависимостей;
- Легкая расширяемость;
- Дамп всех возможных фаз интерпретатора в xml:
  - Лексера;
  - Парсера;
  - Генератора байткода;
- Трассировка программы во время выполнения;

Тестирование

- Тестирование корректности работы управляющих конструкций языка;
- Тестирование корректности работы аллокатора и сборщика мусора;
- Тестирование всего интерпретатора на отсутствие утечек памяти даже при некорректных входных данных (для этого использовался Valgrind);
- Тестирование всего интерпретатора на программе, “имитирующей” сражение двух армий;
- Производительность на уровне Python 3;