Учебный компилятор

Автор: Балышев А.М.
Руководитель: Косарев Д.С.

31 июля 2025 г.

Язык и платформа

- Язык — OCaml
- Хостинг кода — GitHub
- CI/CD — GitHub Actions

Цель и задачи

Цель:
Создание AOT-компилятора вымышленного императивного языка, способного компилировать относительно несложные программы
Задачи:

- Реализация фронтенда
  - Лексический разбор
  - Синтаксический разбор
  - Семантический разбор
- Реализация бэкенда
  - Порождение RISC-V Assembly

О компилируемом языке

- Поддержка базовых императивных конструкций
  - Объявления и присваивания
  - Циклы
  - Ветвления
  - Описание и вызов процедур
- Поддержка типов и областей видимости

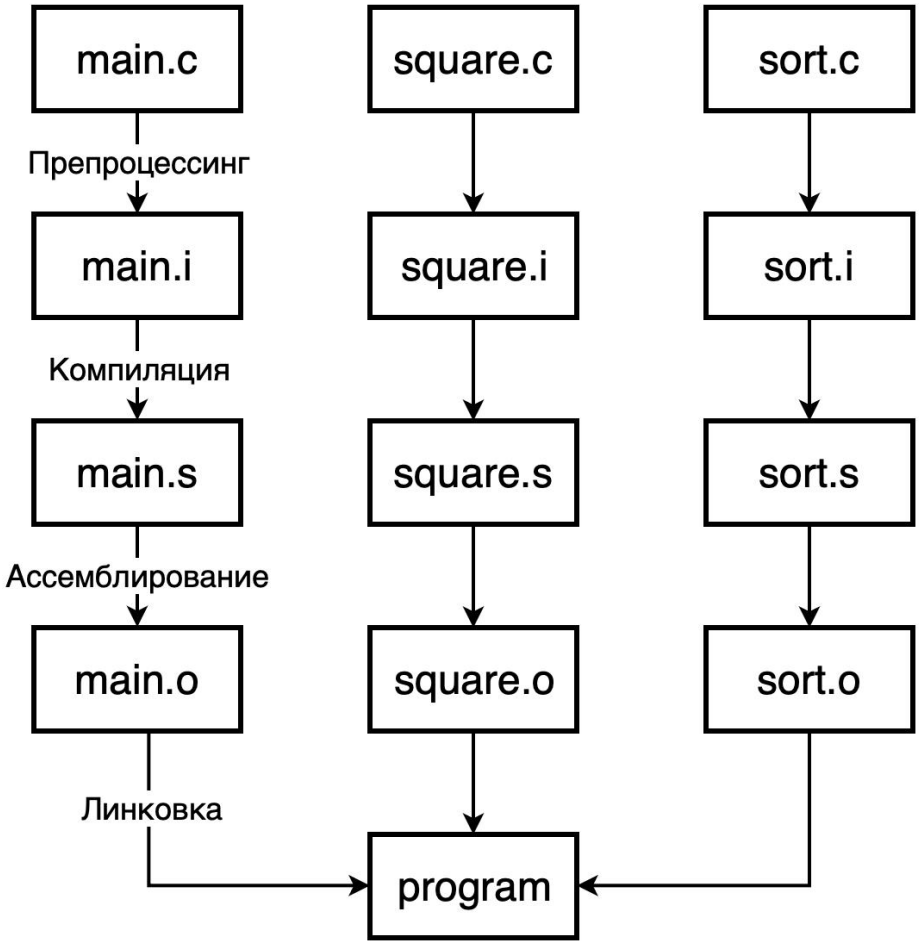
Пример

```
var a b j;

a := 0;
b := 1;
j := 0;

while j < 8 do
  print a;
  b := a + b;
  a := b - a;
  j := j + 1;
done
```

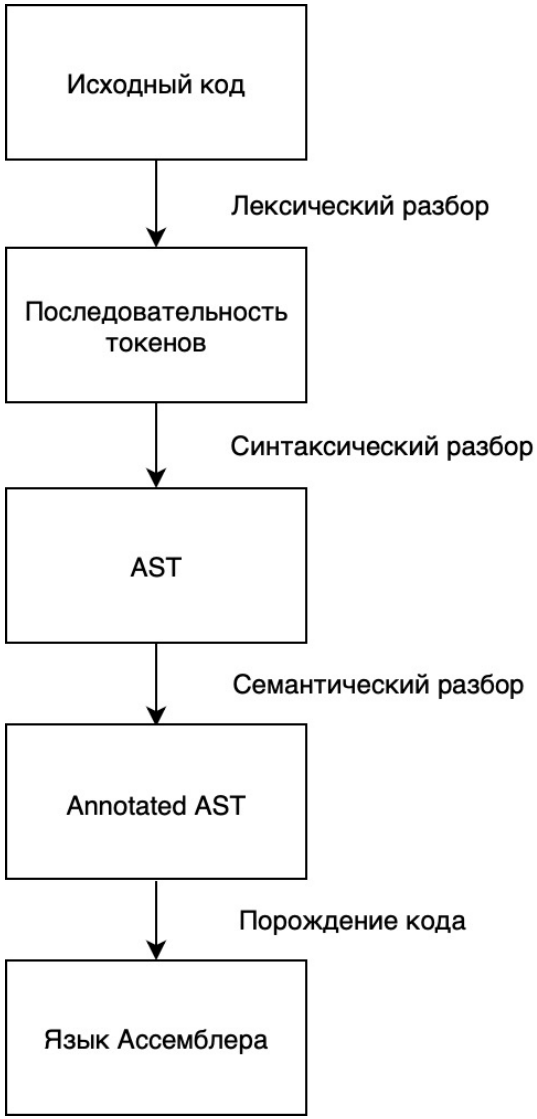
О процессе компиляции

- Каждый этап обрабатывается отдельной функцией
- Функция принимает некоторое представление программы и возвращает новое представление, снабженное дополнительной информацией
- Компиляция осуществляется путём последовательного применения этих функций

Этапы построения исполняемого модуля



Этапы компиляции



Лексический разбор

Задачи:

- Преобразовать исходный код в последовательность токенов
- Выявить лексические ошибки

Сигнатура функции:

```
val tokenize : string -> token list
```

В качестве токенов могут выступать:

- Ключевые слова
  - `while`, `do`, `done`, `if`, `then`, `else`, `fi` ...
- Литералы
  - `Int(123)`, `True`, `False`, `String("hello world")`
- Унарные и бинарные операторы и скобки
- Идентификаторы переменных и функций

Синтаксический разбор

Задачи:

- Преобразовать последовательность токенов в AST
- Выявить синтаксические ошибки

Сигнатура функции:

```
val construct_ast : token list -> program
```

AST

```
type program = statement list

type statement =
| Assignment of string * expression
| While of expression * program
| Ite of expression * program * program
| ...

type expression =
| Var of string
| Int of int
| BinOp of operation * expression * expression
| ...
```

Семантический разбор

Задачи:

- Дополнить AST информацией о типах и областях видимости
- Выявить семантические ошибки

Сигнатура функции:

```
val annotate_ast : program -> typed_program
```

Поддерживаемые на данный момент типы:

- Int
- Bool
- String

Annotated AST

```
type typed_program = (typed_statement * scope) list

type typed_statement =
| Assignment of string * typed_expression
| While of typed_expression * typed_program
| Ite of typed_expression * typed_program * typed_program
| ...

type typed_expression =
| Type_Int of ...
| Type_Bool of ...
| Type_Str of ...
```

Порождение кода

Задача:

- Генерация языка Ассемблера по AST

Сигнатура функции:

```
val generate_assembly : annotated_ast -> string
```

- Порождение кода реализовано для RISC-V 64-bit
- Локальные переменные размещаются в стеке
- ASCIIZ-строки — в куче

Дальнейшие этапы

- Ассемблирование
  - Например, с помощью `riscv64-unknown-elf-as`
  - Результат — объектный модуль
- Линковка
  - Например, с помощью `riscv64-unknown-elf-ld`
  - Результат — исполняемый файл
- Исполнение
  - Или нативно, если архитектура соответствующая
  - Или с помощью эмулятора, например `qemu`

О проблемах и решениях

- Восстановление sp при выходе из scope'а
  - Можно передавать в scope и возвращать из него количество переменных на стеке
- Затенение переменных и функций
  - Можно хранить и искать содержимое scope в порядке объявления
- Метки не должны повторяться
  - Можно генерировать их случайно

Заключение

Использовавшиеся инструменты:

- `ppx_expect`, `ppx_deriving`
- `ocamlformat`
- `riscv64-unknown-elf-as`
- `riscv64-unknown-elf-ld`
- `qemu-riscv64`
- `spike`

Ссылка на репозиторий: https://github.com/psiblvdegod/compiler