Название

Автор

29 июля 2025 г.

Язык и платформа

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

Актуальность

- TODO

Цель

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

Задачи

- Изучение процесса компиляции в современных ЯП
- Подготовка окружения
- Реализация фронтенда
  - Лексический анализ
  - Синтаксический анализ
  - Семантический анализ
- Реализация бекенда
  - Порождение RISC-V Assembly
- Рефакторинг кода и оформление репозитория

Обзор аналогов

Clang:

- TODO??

GCC:

- TODO??

О компилируемом языке

- Инструкции верхнего уровня
- Базовые императивные конструкции
  - Объявления и присваивания
  - Циклы
  - Ветвления
  - Вызов процедур
- Поддержка типов и областей видимости
- Pascal-like синтаксис

О процессе компиляции

- Каждый этап обрабатывается отдельным модулем
- Модуль принимает некоторое представление программы и возвращает новое представление, снабженное дополнительной информацией
- Компиляция представляет собой процесс последовательной передачи результатов вычислений предыдущего модуля в следующий модуль

Лексический анализ

Задачи:

- Преобразовать строку 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 42`, `true`, `false`, `String "hello world"`
- Унарные и бинарные операторы и скобки
- Идентификаторы переменных и функций

Синтаксический анализ

Задачи:

- Преобразовать последовательность токенов в AST
- Выявить синтаксические ошибки

Сигнатура функции:

```
val build_ast : token list -> ast
```

Детали реализации:

- Разбор выражений с помощью рекурсивного спуска
- ???

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 анализ

Задачи:

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

```
val annotate_ast : ast -> annotated_ast
```

Поддерживаемые на данный момент типы:

- Int32
- Bool
- ASCIIZ String

Annotated AST

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

type typed_statement =
| Typed_Assignment of string * typed_expression
| Typed_While of typed_expression * typed_program
| Typed_Ite of typed_expression * typed_program *
    typed_program
| ...

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

Порождение кода

Задача — Преобразование AST в язык Ассемблера
Сигнатура функции:

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

- AST универсально для любой платформы для которой существует компилятор OCaml подходящей версии
- Порождение кода реализовано только для riscv64

???

Результат работы компилятора — код на языке Ассемблера

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 помощью эмулятора, например `qemu`

Использовавшиеся инструменты

- `ppx_expect`, `ppx_deriving`
  - Специфичные для OCaml фреймворки значительно упрощающие тестирование и отладку
- `riscv64-unknown-elf-as`, `riscv64-unknown-elf-ld`
- `qemu-riscv64`, `spike`

Перспективы развития

- Добавление препроцессинга
- Добавление оптимизаций
- Расширение системы типов