Учебный компилятор

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

29 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 бекенда
  - Порождение RISC-V Assembly
- Рефакторинг кода и оформление репозитория

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 отдельной функцией
- Функция принимает некоторое представление программы и возвращает новое представление, снабженное дополнительной информацией
- Компиляция представляет собой последовательную передачу результатов вычислений предыдущей функции в следующую функцию

Лексический разбор

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 разбор

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 разбор

Задачи:

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

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

Поддерживаемые на данный момент типы:

- Int
- 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
```

Порождение кода реализовано только для riscv64

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
  - Нативно, если архитектура соответствующая
  - С помощью эмулятора, например `qemu`

Пример

```
var left right;
left := -7; right := 8;
while left < right do
    var n acc sign;
    n := left;
    acc := 1;
    sign := n < 0;
    while n != 0 do
        acc := acc * n;
        if sign then n := n + 1;
        else  n := n - 1;
        fi
    done
    left := left + 1;
    printn acc;
done
```

О проблемах и решениях

- При выделении места на стеке для локальных переменных при выходе из области видимости нужно восстановить sp
  - Можно хранить текущее количество переменных
- Метки для условных/безусловных переходов не должны повторяться
  - Можно генерировать их случайно

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`