Компиляторы. Введение

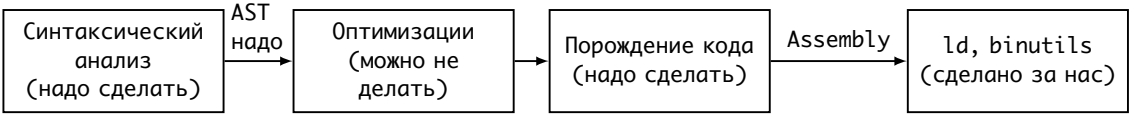
Косарев Дмитрий

матмех

июль 2024 г.

Дата сборки: 8 июля 2024 г.

Картинка про архитектуру компиляторов



- Синтаксический анализатор (парсер) можно просто брать и делать
- Порождение кода — просто, если разбираться в ассемблере
- Ассемблер RISC-V 64 — тут придется повозиться, чтобы понять что да как

Начинать разбирательство нужно с AST (дерево абстрактного синтаксиса, т.е представление программы), затем в любом порядке парсер и ассемблер, потом порождение кода

Язык, который будем компилировать

Две синтаксические категории
Выражения (expressions):

- Константы (целочисленные)
- Бинарные операции ($+$, $-$, $\times$, /)

Инструкции (или операторы, statements):

- Присваивание
- Ветвления
- Цикл while

Потом можно будет расширять...

Факториал

```
acc:=1; n:=6;
while n>1 do
  acc:=acc*n;
  n:=n-1;
done
```

Фибоначчи

```
a:=0; b:=1; n:=5;
while n>1 do
  b:=a+b;
  a:=b-a;
  n:=n-1;
done
```

Чего в язык не включаем

Когда-нибудь потом:

- Функции — когда изучим ассемблер
- Строки, массивы
- Проверка типизации

Вообще, компилятор лучше разрабатывать слоями

- Представление, парсер, порождение кода для базового языка выражение
- Представление, парсер, порождение кода для одного расширения операторами
- ... для другого расширения

Во что это должно вылиться

1. Скрещивать OCaml и Rust [4, 5, 3].
2. Добавлять топовые оптимизации [1].
3. Ускорять OCaml под RISC V (например, интринсиками [6]).
4. Делать свой OCaml, где можно грабить корованы
5. прикручивать к OCaml порождение кода с помощью LLVM

— Бери сложную тему для курсача — интересно будет.
— Ну да, интересно. Интересно, сколько я протяну.

[0] https://www.meme-arsenal.com/create/template/3333768

Представление программы

Дерево абстрактного синтаксиса (англ. abstract syntax tree, AST)

- древовидное представление
- без скобок, комментариев — из-за этого называется «абстрактным»

По сути большое описание альтернатив: какие бывают выражения, операторы
Делается по-разному, в зависимости от языка программирования

Представление в функциональном стиле (1/3 OCaml, Rust)

OCaml

```
type oper = Plus | Multiply | Divide
type expr =
    | Const of int
    | Binop of oper * expr * expr
```

Rust

```
enum Oper { Add, Mul, Sub }
enum Expr {
    Const(i64),
    Binop(Oper, Box<Expr>,  Box<Expr>)
}
```

Представление в ООР стиле (2/3 C++)

```
class Expr {
};
class EConst : public Expr {
    int val;
};
enum Operator { ADD, SUB, MUL };
class EBinop : public Expr {
    Expr *left;
    Expr *right;
    Operator op;
};
```

Многословное представление (3/3 C)

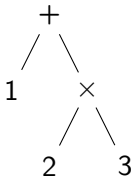
```c
typedef struct AST AST; // Forward reference

struct AST {
  enum { AST_NUMBER, AST_ADD, AST_MUL } tag;
  union {
    struct AST_NUMBER { int number; } AST_NUMBER;
    struct AST_ADD { AST *left; AST *right; } AST_ADD;
    struct AST_MUL { AST *left; AST *right; } AST_MUL;
  } data;
};
```

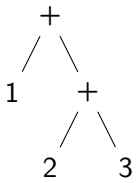
Как это использовать: [2]

Примеры AST. Выражения с приоритетами операций

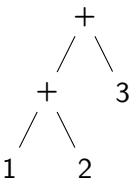
$1 + 2 \times 3$



$1 + (2 + 3)$

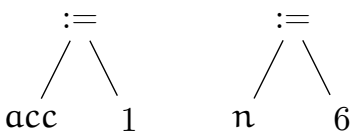


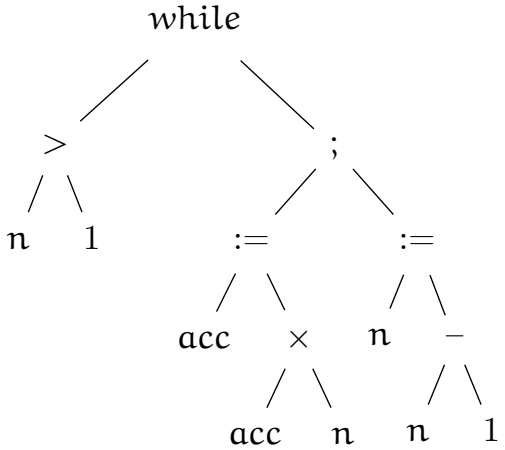
$(1 + 2) + 3$



Факториал

```
acc:=1; n:=6;
while n>1 do
  acc:=acc*n;
  n:=n-1;
done
```





Рекурсивный спуск

Состояние

- `text` — строка ASCII, которую разбираем
- `length` — длина строки, чтобы не пересчитывать
- `pos` — текущая позиция

Наши функции-парсеры, будут принимать аргументы, и возвращать либо успешный результат, либо ошибку

Рекурсивный спуск. Примеры

Пустые символы (пробелы, переводы строк и т.п.)

```
let ws () =
  while pos<length && is_whitespace(text[pos]) do
    incr pos
  done
```

В примерах ниже в некоторых местах должны будут стоять парсеры пробелов, я их буду забывать

Рекурсивный спуск. Примеры. Целочисленные константы

```
let econst () =
  let acc = ""
  while pos < length && is_digit(text[pos]) do
    acc += text[pos]
    incr pos
  done;
  if acc.length > 0 then
    return success and EConst (int_of_string (acc))
  else return error
```

Ошибка, если цифр нет.

Рекурсивный спуск. Примеры

Бывают ключевые слова (if, while и т.д.), и идентификаторы. Идентификакторы (для простоты) — это последовательность строчных букв, не являющаяся ключевым словом

```
let ident_or_keyword () =
  let acc = ""
  while pos<length && is_alpha(text[pos]) do
    acc += text[pos]
    incr pos
  done;
  if acc.length > 0 then return success and acc
  else return error

let ident () =
  let s = ident_or_keyword ()
  if is_keyword(s) then  return error
  else return success and x
```

Рекурсивный спуск. Примеры

Бывают ключевые слова (if, while и т.д.), и идентификаторы. Идентификаторы (для простоты) — это последовательность строчных букв, не являющаяся ключевым словом

```
let ident_or_keyword () =
  let acc = ""
  while pos<length && is_alpha(text[pos]) do
    acc += text[pos]
    incr pos
  done
  if acc.length > 0 then return success and acc
  else return error

let ident () =
  let s = ident_or_keyword ()
  if is_keyword(s) then  return error
  else return success(x)
```

Рекурсивный спуск. Выражения (самая сложная часть, 1/?)

Давайте упомянем формализм грамматик...

Можно так...

$$
\begin{aligned}
\langle \text{expr} \rangle &::= \langle \text{const} \rangle \mid \\
&\quad\ \langle \text{const} \rangle \langle \text{oper} \rangle \langle \text{expr} \rangle \\
\langle \text{oper} \rangle &::= + \mid - \mid \times \mid /
\end{aligned}
$$

... или так

$$
\begin{aligned}
\langle \text{expr} \rangle &::= \langle \text{const} \rangle \langle \text{extra} \rangle \\
\langle \text{extra} \rangle &::= \epsilon \mid \langle \text{oper} \rangle \langle \text{const} \rangle \langle \text{extra} \rangle \\
\langle \text{oper} \rangle &::= + \mid - \mid \times \mid /
\end{aligned}
$$

Вот так делать не надо

$$
\begin{aligned}
\langle \text{expr} \rangle &::= \langle \text{expr} \rangle \langle \text{oper} \rangle \langle \text{const} \rangle \mid \\
&\quad\ \langle \text{const} \rangle
\end{aligned}
$$

Из-за левой рекурсии парсер будет зависать.

Идея заглядывания вперед (англ. look ahead)

1. **Запоминаем**, где в строке мы находимся
2. Разбираем то, что дальше в строке написано и получаем результат
3. **Возвращаем позицию** в строке на исходную

```
let expr () =
  let head = number()
  let pos1 = pos      (* 1 *)
  let op = oper()     (* 2 *)
  if op.success
  then return combine (head, op, expr())
  else
    rollback pos1     (* 3 *)
    return head
```

Рекурсивный спуск. Добавляем умножение

Без скобок

$$
\begin{aligned}
\langle \text{expr} \rangle &::= \langle \text{prod} \rangle + \langle \text{expr} \rangle \mid \\
&\quad\ \langle \text{prod} \rangle \\
\langle \text{prod} \rangle &::= \langle \text{const} \rangle \times \langle \text{prod} \rangle \mid \\
&\quad\ \langle \text{const} \rangle
\end{aligned}
$$

Со скобками

$$
\begin{aligned}
\langle \text{expr} \rangle &::= (\langle \text{expr} \rangle) \mid \\
&\quad\ \langle \text{prod} \rangle + \langle \text{expr} \rangle \mid \\
&\quad\ \langle \text{prod} \rangle \\
\langle \text{prod} \rangle &::= (\langle \text{expr} \rangle) \mid \\
&\quad\ \langle \text{const} \rangle \times \langle \text{prod} \rangle \mid \\
&\quad\ \langle \text{const} \rangle
\end{aligned}
$$

А далее это можно усложнять: учитывать ассоциативность, добавлять меньшие приоритеты and, or...

Если работают выражения, то statement добавить просто

$$
\begin{aligned}
\langle \text{stmt} \rangle &::= \langle \text{ident} \rangle \texttt{ := } \langle \text{expr} \rangle \texttt{;} \mid \\
&\quad \texttt{if } \langle \text{expr} \rangle \texttt{ then } \langle \text{stmts} \rangle \texttt{ else } \langle \text{stmts} \rangle \texttt{ fi} \mid \\
&\quad \texttt{while } \langle \text{expr} \rangle \texttt{ do } \langle \text{stmts} \rangle \texttt{ done} \\
\langle \text{stmts} \rangle &::= \epsilon \mid \\
&\quad \langle \text{stmt} \rangle \langle \text{stmts} \rangle \\
\langle \text{program} \rangle &::= \langle \text{stmts} \rangle
\end{aligned}
$$

Ссылки I

[1] Pierre Chambart, Vincent Laviron и Mark Shinwell. *Efficient OCaml compilation with Flambda 2*. URL: https://icfp23.sigplan.org/details/ocaml-2023-papers/8/Efficient-OCaml-compilation-with-Flambda-2.

[2] Vladimir Keleshev. *Abstract Syntax Tree: an Example in C*. https://keleshev.com/abstract-syntax-tree-an-example-in-c. 2022.

[3] Anton et al. Lorenzen. *Oxidizing OCaml with Modal Memory Management*. URL: https://homepages.inf.ed.ac.uk/slindley/papers/mode-inference-draft-feb2024.pdf.

[4] Max Slater. *Oxidizing OCaml: Locality*. URL: https://blog.janestreet.com/oxidizing-ocaml-locality.

[5] Max Slater. *Oxidizing OCaml: Rust-Style Ownership*. URL: https://blog.janestreet.com/oxidizing-ocaml-ownership.

Ссылки II

[6] Азат Габдрахманов. *Интринсики RISC-V для компилятора OCaml*. URL: https://se.math.spbu.ru/thesis/slides/Gabdrahmanov_Azat_Rajnurovich_Autumn_practice_3rd_year_2023_slides.pdf.