Компиляторы. Введение

Косарев Дмитрий

матмех

июль 2025 г.

Дата сборки: 7 июля 2025 г.

В этих слайдах

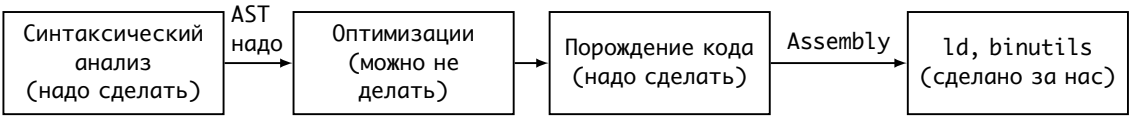
1. Краткая архитектура

2. Синтаксический анализ (парсинг)

3. Тестирование парсера

Оглавление

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 можно компилировать (а ля ALGOL)

Вы можете смело компилировать то, на чем пишете

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

Оглавление

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)

Самый краткий вариант

OCaml

```
type oper = Plus | Aster | Slash
type expr =
    | Const of int
    | Binop of oper * expr * expr
```

Rust

```
enum Oper { Plus, Aster, Slash }
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)

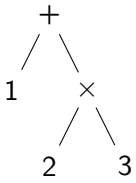
```
typedef struct AST AST; // Forward reference

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

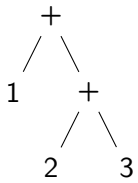
Как это использовать: [2]

Примеры AST. Выражения с приоритетами операций

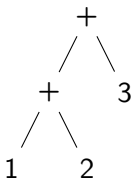
$1 + 2 \times 3$



$1 + (2 + 3)$



$(1 + 2) + 3$

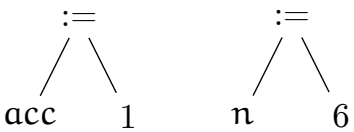


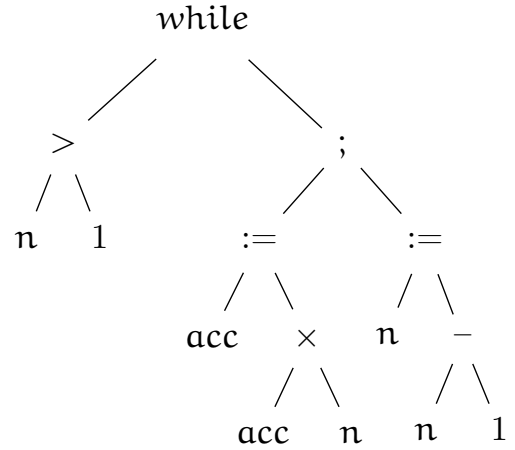
«Algol», где присутствуют и statement, и expression

А также, Pascal, C, Java, Kotlin, Go, Rust...

Факториал

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





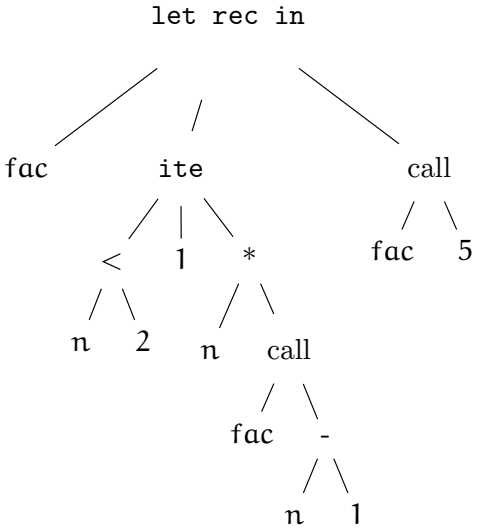
«ML», где только выражения (expression)

ML, Haskell, OCaml, Scala (?)

```
let rec fac n =
  if n<2 then 1
  else n * fac (n-1)
in
fac 5
```

Упражнение

Что я забыл?



Рекурсивный спуск

Стандарт для большинства компиляторов (никогда не догадаетесь почему)

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

Далее код будет из [1]

Рекурсивный спуск. Примеры

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

```
bool is_ws(char c) {
  switch (c) {
  case '\n':
  case ' ':    return true;
  }
  return false;
}
void ws() {
  while (pos < length && is_ws(text[pos]))
    pos++;
}
```

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

Рекурсивный спуск. Примеры. Конкретные строки

можно использовать для ключевых слов

```
bool pstring(char *str) {
  unsigned len = strlen(str);
  for (auto i = 0; i < len; ++i)
    if (pos < length && text[pos] == str[i]) {
      pos++;
    } else {
      return false;
    }
  return true;
}
```

Рекурсивный спуск. Примеры. Целочисленные константы

```
bool econst(AST &ast) {
  auto oldpos = pos, left = pos, right = pos, acc = 0;
  if (pos < length && is_digit(text[pos])) {
    acc = text[pos] - '0';
    pos++; right++;
  } else return false;

  if (right > left)
    while (pos < length && is_digit(text[pos])) {
      acc = acc * 10 + text[pos] - '0';
      pos++; right++;
    }
  if (right > left) { /* TODO: ast */ return true;
  } else { pos = oldpos;  return false;  }
}
```

Рекурсивный спуск. Примеры

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

```
bool eident(AST &ast) {
  unsigned left, right;
  if (ident(left, right)) {
    // TODO: check keywords
    assert(right > left);
    char *str = new char[right - left + 1];
    memset(str, '\0', right - left + 1);
    memcpy(str, text + left, right - left); /* TODO: ast */
    return true;
  }
  return false;
}
bool ident(unsigned, unsigned);  // exercise
```

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

Давайте упомянем формализм грамматик... Но я его не люблю

Можно так...

$$
\begin{aligned}
\langle \text{expr} \rangle &::= \langle \text{const} \rangle \langle \text{oper} \rangle \langle \text{expr} \rangle \mid \\
&\quad\ \langle \text{const} \rangle \\
\langle \text{oper} \rangle &::= \texttt{+} \mid - \mid \times \mid \texttt{/} \\
\langle \text{const} \rangle &::= \langle \text{digit} \rangle \langle \text{const} \rangle \mid \langle \text{digit} \rangle \\
\langle \text{digit} \rangle &::= \texttt{0} \mid \texttt{1} \mid \texttt{...} \mid \texttt{9}
\end{aligned}
$$

... или так

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

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

```
bool expr () {
  AST left, right; char* operand;
  bool success = number(left) // if can't -- fail
  auto pos1 = getCurPosition()     // 1
  oper(operand)    // 2.
  if (success1 && success2 && ...) {
    expr(right)
    return true; // makeAST(left, op, right)
  } else {
    rollback(pos1);   // 3
    return true; // left is our result
  }
}
```

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

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

Здесь забыты константы и арифметика

Вот то же самое для языка без statement

$$
\begin{aligned}
\langle \text{expr} \rangle &::= \texttt{(}\ \langle \text{expr} \rangle\ \langle \text{expr} \rangle\ \texttt{)}\ \mid \\
&\quad \langle \text{ident} \rangle\ \mid \\
&\quad \texttt{(}\lambda\langle \text{ident} \rangle\texttt{.}\langle \text{expr} \rangle\texttt{)} \\
&\quad \langle \text{const} \rangle\ \mid \\
&\quad \texttt{(}\ \langle \text{expr} \rangle\ \texttt{+}\ \langle \text{expr} \rangle\ \texttt{)}\ \mid \ \ldots \\
&\quad \texttt{if}\ \langle \text{expr} \rangle\ \texttt{then}\ \langle \text{expr} \rangle\ \texttt{else}\ \langle \text{expr} \rangle\ \mid \\
&\quad \texttt{let rec}\ \langle \text{ident} \rangle\ \texttt{=}\ \langle \text{expr} \rangle\ \texttt{in}\ \langle \text{expr} \rangle \\
\langle \text{program} \rangle &::= \langle \text{expr} \rangle
\end{aligned}
$$

Оглавление

Тестирование парсера

Оставлено до лучших времён

Ссылки I

[1] *C parser of expressions*. 2025. URL: https://github.com/Kakadu/rukaml/blob/ss/lib/ParseSIMD/parser_in_c.cpp.

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