

1 Первая домашка

Язык программирования (ЯП) — множество программ на этом языке. ЯП — частный случай всех языков. Любой язык состоит из синтаксиста (правила построения предложений), семантики (смысл предложений) и прагматики (взаимодействие с носителем).

В языках программирования разделяют **абстрактный и конкретный синтаксис**.

Абстрактный синтаксис языка выражений:

$$\begin{aligned}\mathcal{X} &= \{x, y, z, \dots\} \\ \otimes &= \{+, -, \times, /, \%, <, \leq, >, \geq, =, \neq, \vee, \wedge\} \\ \mathcal{E} &= \mathcal{X} \\ &\quad \mathbb{N} \\ &\quad \mathcal{E} \otimes \mathcal{E}\end{aligned}$$

Задано множество переменных, множество бинарных операций и множество **абстрактных синтаксических деревьев** (АСД).

АСД является:

- изолированным узлом, который представляет переменную или число;
- узлом бинарной операции с двумя дочерними деревьями.

Следовательно, дерево определяется по индукции. Дерево упорядочено.



Множество АСД является языком, а каждая программа определяется АСД. Пример АСД:



Пример конкретного синтаксиста: $(a + b) * c$. Чтобы абстрактный синтаксис выразить в конкретный потребуются дополнительные понятия (в данном случае старшинство операций).

Семантика — это всюду определенное отображение языка в семантический домен:

$$\llbracket \bullet \rrbracket_{\mathcal{L}} : \mathcal{L} \mapsto \mathcal{D}$$

Функциональный семантический домен — множество функций.

Введем **функцию состояния** $\sigma : \mathcal{X} \rightarrow \mathbb{Z}$, которая связывает переменные с их значениями.

Рассмотрим **операционную семантику большого шага**. Она определяется тернарным отношением:

$$\Rightarrow_{\mathcal{E}} \subseteq \Sigma \times \mathcal{E} \times \mathbb{Z}.$$

Запись $\sigma \xRightarrow{\mathcal{E}} n$ означает, что вычисление выражения e в состоянии σ дает значение n .

Три семантических правила:

$$\begin{array}{ll} \frac{n \in \mathbb{N}}{\sigma \xRightarrow{\mathcal{E}} n} & [\text{CONST}] \\ \frac{x \in \mathcal{X}}{\sigma \xrightarrow{x}_{\mathcal{E}} \sigma x} & [\text{VAR}] \\ \frac{\sigma \xRightarrow{l}_{\mathcal{E}} x, \sigma \xRightarrow{r}_{\mathcal{E}} y}{\sigma \xRightarrow{l \otimes r}_{\mathcal{E}} x \oplus y} & [\text{BINOP}] \end{array}$$

В случае с переменной мы должны извлечь из переменной число, чтобы получить результат.

Линия отделяет **посылку** (правило сверху) от **заключения** (правило снизу). К заключению можно приступить только если выполнена посылка. Если у правила отсутствует посылка, то оно является **аксиомой**.

Базовые операции:

\otimes	\oplus in \mathcal{AM}^a
+	+
—	—
\times	*
/	/
%	%
<	<
>	>
\leq	\leq
\geq	\geq
=	=
\neq	\neq
\wedge	&&
\vee	!!

Пример программы: $1 \wedge (1/0)$. Пример описания:

$$\frac{\frac{\frac{\wedge}{\Rightarrow 1} \quad \frac{\vee}{\Rightarrow 0}}{\wedge \vee (\wedge / \vee)}}{\wedge \vee (\wedge / \vee)}$$

Результатом будет неопределенность, так как результат деления на ноль не определен.

Мы рассмотрели **строгую (strict) семантику**. Функция называется неопределенной, если функция от неопределенности есть неопределенность.

Добавим уточнение, что бинарная операция $\neq \wedge \mid \vee$ и добавим два новых правила, которые сделают семантику нестрогой по второму аргументу:

$$\frac{c \stackrel{e}{\Rightarrow} 1}{c \stackrel{e \vee r}{\Rightarrow} 1} \quad \frac{c \stackrel{e}{\Rightarrow} z+1, d \stackrel{e}{\Rightarrow} z'}{c \stackrel{e \vee r}{\Rightarrow} z \parallel z'}$$

Если левый аргумент — единица, то результат будет единицей. Если левый аргумент не равен единице, а правый аргумент — целое число, то результат вычисляется из этих двух чисел.

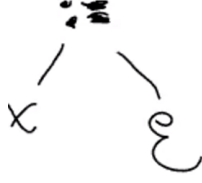
Обобщение правил $\Rightarrow_{\mathcal{E}}$ в семантику $\llbracket \bullet \rrbracket_{\mathcal{E}}$:

$$\frac{\sigma \stackrel{e}{\Rightarrow}_{\mathcal{E}} n}{\llbracket e \rrbracket_{\mathcal{E}} \sigma = n}$$

Язык состоит из множества синтаксических категорий. Существует несколько синтаксических категорий. Категорию выражений (expression) мы рассмотрели. Следующая категория — операторов (statement). Рассмотрим язык операторов:

$\mathcal{S} =$ **skip**
 $\mathcal{X} := \mathcal{E}$
read(\mathcal{X})
write(\mathcal{E})
 $\mathcal{S}; \mathcal{S}$

Второе правило определяет операцию присвоения, где слева находится переменная, справа — выражение (рассмотрено ранее).



Последнее правило определяет композицию операторов.

Определим семантику этого языка. Пусть наш язык описывает программы, которые принимают строки целых чисел и возвращают строки целых чисел:

$$\llbracket \bullet \rrbracket : \mathcal{S} \mapsto \mathbb{Z}^* \rightarrow \mathbb{Z}^*$$

Семантика большого шага связывает две конфигурации и программу:

$$\Rightarrow_{\mathcal{S}} \subseteq \mathcal{C} \times \mathcal{S} \times \mathcal{C}.$$

Значит, что при запуске программы в одной конфигурации, через несколько шагов получаем другую конфигурацию. Семантика большого шага так называется, так как при переходе выполняется цепочка шагов.

Конфигурация — это пара состояния и мира:

$$\mathcal{C} : \Sigma \times \mathcal{W}$$

Мир — это инкапсуляция входного и выходного потока:

$$\mathcal{W} : \mathbb{Z}^* \times \mathbb{Z}^*$$

Определим операции:

$$\begin{aligned} \mathbf{read} \langle xi, o \rangle &= \langle x, \langle i, o \rangle \rangle \\ \mathbf{write} \, x \langle i, o \rangle &= \langle i, ox \rangle \\ \mathbf{out} \langle i, o \rangle &= o \end{aligned}$$

Здесь $xi \in \mathbb{Z}^*$ — конкатенация слов x и i . Создает новый мир, где входной поток определяется x , а выходной прошлым миром. Запись добавляет x в конец выходного потока. Вывод выводит выходной поток.

$$\begin{array}{c}
c \xRightarrow{\text{skip}}_{\mathcal{S}} c \quad [\text{SKIP}] \\
\langle \sigma, \omega \rangle \xRightarrow{x := e}_{\mathcal{S}} \langle \sigma[x \leftarrow \llbracket e \rrbracket_{\mathcal{E}} \sigma], \omega \rangle \quad [\text{ASSIGN}] \\
\frac{\langle z, \omega' \rangle = \text{read } \omega}{\langle \sigma, \omega \rangle \xRightarrow{\text{read}(x)}_{\mathcal{S}} \langle \sigma[x \leftarrow z], \omega' \rangle} \quad [\text{READ}] \\
\langle \sigma, \omega \rangle \xRightarrow{\text{write}(e)}_{\mathcal{S}} \langle \sigma, \text{write}(\llbracket e \rrbracket_{\mathcal{E}} \sigma) \omega \rangle \quad [\text{WRITE}] \\
\frac{c_1 \xRightarrow{S_1}_{\mathcal{S}} c' \quad c' \xRightarrow{S_2}_{\mathcal{S}} c_2}{c_1 \xRightarrow{S_1; S_2}_{\mathcal{S}} c_2} \quad [\text{SEQ}]
\end{array}$$

Во время присвоения конфигурация явно представляется в виде пары состояния и мира $\langle \sigma, \omega \rangle$. В результате переменной x в состоянии σ присваивается результат выполнения выражения, а мир не изменяется.

Операционная семантика языка определяет эталонный интерпретатор. Интерпретатор — это программа, которая принимает программу на данном языке и для любого входа возвращает выход этой программы. Семантика интерпретатора:

$$\begin{aligned}
\llbracket \text{int} \rrbracket : \mathcal{L} &\rightarrow I \rightarrow O \\
\llbracket \text{int} \rrbracket_{\mathcal{L}}(p, x) &= \llbracket p \rrbracket_{\mathcal{M}}(x)
\end{aligned}$$

Обычно скобки не ставят, чтобы запись не была перегружена разными символами и читалась легче.

Обобщим семантику оператора и определим $\llbracket \bullet \rrbracket_{\mathcal{S}}$:

$$\frac{\langle \Lambda, \langle i, \epsilon \rangle \rangle \xRightarrow{S}_{\mathcal{S}} \langle \sigma, \omega \rangle}{\llbracket S \rrbracket_{\mathcal{S}} i = \text{out } \omega}$$

Семантика принимает числа i , после чего из когфигурации, где сосояние пусто (ни одна переменна не определена), выполняется переход по соответствующему правилу в конфигурацию с каким-то состоянием и миром. Результатом является выходной поток полученного мира.

Наш компилятор является многопроходным. Сначала получаем программу в виде синтаксического дерева, затем интерпретируем ее в соответствии с семантикой большого шага. Далее мы компилируем программу в стековую машину, а из стековой машины можем выполнить компиляцию в x86.

Фактически стековая машина является языком программирования со своей семантикой.

Синтаксис стековой машины состоит из двух синтаксических категорий: инструкции (instructions) и программы (programs)

$$\begin{aligned}
\mathcal{I} &= \text{BINOP} \otimes \\
&\quad \text{CONST } \mathbb{N} \\
&\quad \text{READ} \\
&\quad \text{WRITE} \\
&\quad \text{LD } \mathcal{X} \\
&\quad \text{ST } \mathcal{X} \\
\mathcal{P} &= \varepsilon \\
&\quad \mathcal{I} \mathcal{P}
\end{aligned}$$

Программа стековой машины является списком (возможно пустым) инструкций.

Конфигурация стековой машины включает стек. Множество конфигураций можно определить следующим образом:

$$\mathcal{C}_{SM} = \mathbb{Z}^* \times \mathcal{C}$$

Правила семантики большого шага для СМ:

$$\begin{aligned}
c &\xRightarrow{\varepsilon}_{\mathcal{SM}} c && [\text{STOP}_{SM}] \\
\frac{\langle (x \oplus y)s, c \rangle \xRightarrow{p}_{\mathcal{SM}} c'}{\langle yxs, c \rangle \xRightarrow{[\text{BINOP } \otimes]p}_{\mathcal{SM}} c'} &&& [\text{BINOP}_{SM}] \\
\frac{\langle zs, c \rangle \xRightarrow{p}_{\mathcal{SM}} c'}{\langle s, c \rangle \xRightarrow{[\text{CONST } z]p}_{\mathcal{SM}} c'} &&& [\text{CONST}_{SM}] \\
\frac{\langle z, \omega' \rangle = \mathbf{read } \omega, \langle zs, \langle \sigma, \omega' \rangle \rangle \xRightarrow{p}_{\mathcal{SM}} c'}{\langle s, \langle \sigma, \omega \rangle \rangle \xRightarrow{\text{READ } p}_{\mathcal{SM}} c'} &&& [\text{READ}_{SM}] \\
\frac{\langle s, \langle \sigma, \mathbf{write } z \omega \rangle \rangle \xRightarrow{p}_{\mathcal{SM}} c'}{\langle zs, \langle \sigma, \omega \rangle \rangle \xRightarrow{\text{WRITE } p}_{\mathcal{SM}} c'} &&& [\text{WRITE}_{SM}] \\
\frac{\langle (\sigma x)s, \langle \sigma, \omega \rangle \rangle \xRightarrow{p}_{\mathcal{SM}} c'}{\langle s, \langle \sigma, \omega \rangle \rangle \xRightarrow{[\text{LD } x]p}_{\mathcal{SM}} c'} &&& [\text{LD}_{SM}] \\
\frac{\langle s, \langle \sigma[x \leftarrow z], \omega \rangle \rangle \xRightarrow{p}_{\mathcal{SM}} c'}{\langle zs, \langle \sigma, \omega \rangle \rangle \xRightarrow{[\text{ST } x]p}_{\mathcal{SM}} c'} &&& [\text{ST}_{SM}]
\end{aligned}$$

p на стрелке означает выполнение оставшейся части программы.

LD кладет значение переменной (σx) в стек. ST кладет значение из стека z в переменную x ($\sigma[x \leftarrow z]$).

Семантика СМ:

$$\frac{\langle \varepsilon, \langle \Lambda, \langle i, \varepsilon \rangle \rangle \rangle \xRightarrow{p}_{\mathcal{SM}} \langle s, \langle \sigma, \omega \rangle \rangle}{\llbracket p \rrbracket_{\mathcal{SM}} i = \mathbf{out } \omega}$$

Компилятор из языка операторов в стековую машину:

$$\llbracket \bullet \rrbracket^{comp} : \mathcal{S} \mapsto \mathcal{P}$$

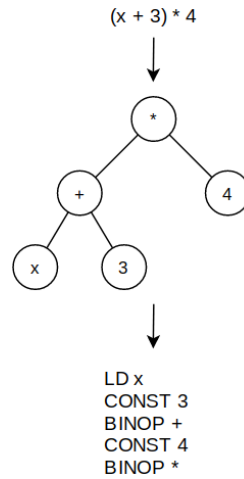
Компилятор — это функция, которая отображает программу на одном языке в программу на другом, сохраняя при этом семантику.

Компилятор можно рассматривать как семантику языка программирования. У компилятора в качестве семантического домена выступает другой язык. Таким образом, у языка может быть больше одной семантики.

Опишем семантику в виде. Просто потому что так проще. Пусть исходный язык состоит из двух синтаксических категорий: выражения и операторы. Компилятор будет состоять из двух функций, которые генерируют код для каждой категории. Компилятор выражений (получает выражение, возвращает последовательность инструкций):

$$\begin{aligned} \llbracket x \rrbracket_{\mathcal{E}}^{comp} &= [\text{LD } x] \\ \llbracket n \rrbracket_{\mathcal{E}}^{comp} &= [\text{CONST } n] \\ \llbracket A \otimes B \rrbracket_{\mathcal{E}}^{comp} &= \llbracket A \rrbracket_{\mathcal{E}}^{comp} \llbracket B \rrbracket_{\mathcal{E}}^{comp} [\text{BINOP } \otimes] \end{aligned}$$

Пример компиляции:



Компилятор операторов:

$$\begin{aligned} \llbracket x := e \rrbracket^{comp} &= \llbracket e \rrbracket_{\mathcal{E}}^{comp} [\text{ST } x] \\ \llbracket \text{read}(x) \rrbracket^{comp} &= [\text{READ}] [\text{ST } x] \\ \llbracket \text{write}(e) \rrbracket^{comp} &= \llbracket e \rrbracket_{\mathcal{E}}^{comp} [\text{WRITE}] \\ \llbracket S_1; S_2 \rrbracket^{comp} &= \llbracket S_1 \rrbracket^{comp} \llbracket S_2 \rrbracket^{comp} \end{aligned}$$

$$\llbracket \text{skip} \rrbracket^{comp} = \epsilon$$

В случае с присвоением мы при помощи компилятора для выражений получаем значение, затем кладем его в переменную.

Ветка с домашкой начинается на A01. Если в корне проекта выполнить `make`, то начнут выполняться регрессионные тесты. Чтобы их пройти нужно

<code>src/World.lama</code>	Реализация семантики мира
<code>src/State.lama</code>	Реализация состояния
<code>src/Expr.lama,</code> <code>src/Stmt.lama,</code>	Реализация правил и компиляции. Нужно дописать функции, которые возвращают <code>failure</code>
<code>src/SM.lama</code>	

В компиляторе лучше использовать буфер для списка инструкций

Делать пул реквест в тот же бранч. Сначала убедиться, что тесты работают локально. Если его отклонили, то все ок. В табличке результаты. Одна ветка — одно ДЗ. В комментариях к пул реквесту указать кто я.

Установка:

- Ubuntu 18 с 4 ГиБ памяти,
- пакетный менеджер OPAM,
- язык программирования OCaml 4.10.1.

```
$ vagrant init ubuntu/bionic64
$ vagrant up
$ vagrant ssh

# apt-get update
# apt-get -y upgrade
# apt-get -y install gcc-multilib make m4 unzip bubblewrap gdb

# sh <(curl -sL https://raw.githubusercontent.com/ocaml/opam/master/shell/install.sh)
$ opam init -y
$ eval $(opam env)
$ opam switch create lama ocaml-variants.4.10.1+fp+flambda -y
$ eval $(opam env)
$ opam pin add Lama https://github.com/JetBrains-Research/Lama-devel.git\#1.10+
  ocaml4.10 --no-action -y
$ opam depext Lama -y
$ opam install Lama -y
```

2 Вторая домашка