1 Первая домашка

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

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

Абстрактный синтаксис языка выражений:

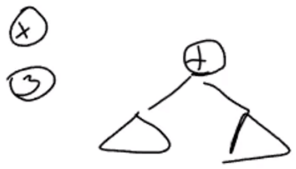
$$\begin{array}{rcl} \mathscr{X} & = & \{x, y, z, \ldots\} \\ \otimes & = & \{+, -, \times, /, \%, <, \leq, >, \geq, =, \neq, \vee, \wedge\} \\ \mathscr{E} & = & \mathscr{X} \\ & & \mathbb{N} \\ & & \mathscr{E} \otimes \mathscr{E} \end{array}$$

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

АСД является:

- изолированным узлом, который помечен переменной или числом;
- узлом бинарной операции с двумя дочерними деревьями.

Следовательно, дерево упорядочено и определяется по индукции (базой является первый случай).



Множество АСД определяет язык, а каждая программа определяется АСД. Пример АСД:

Пример конкретного синтаксиса: `(a + b) * c`. Чтобы абстрактный синтаксис выразить в конкретный потребуются дополнительные понятния (в данном случае старшинство операций).

Семантика — это всюду определенное отображение языка в семантический домен:

$$[\![\bullet]\!]_{\mathscr{L}} : \mathscr{L} \mapsto \mathscr{D}$$

Функциональный семантический домен — множество функций.

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

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

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

Запись $\sigma \overset{e}{\Rightarrow}_{\mathscr{E}} n$ означает, что вычисление выражения e в состоянии σ дает значение n.

Три семантических правила:

$$\frac{n \in \mathbb{N}}{\sigma \overset{n}{\Longrightarrow}_{\mathscr{E}} n} \quad [\textsc{Const}]$$

$$\frac{x \in \mathscr{X}}{\sigma \overset{x}{\Longrightarrow}_{\mathscr{E}} \sigma\, x} \quad [\textsc{Var}]$$

$$\frac{\sigma \overset{l}{\Longrightarrow}_{\mathscr{E}} x,\ \sigma \overset{r}{\Longrightarrow}_{\mathscr{E}} y}{\sigma \overset{l \otimes r}{\Longrightarrow}_{\mathscr{E}} x \oplus y} \quad [\textsc{Binop}]$$

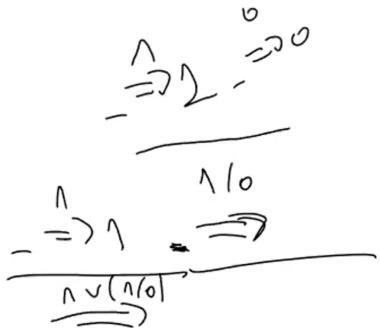
В случае с переменной мы должны извлечь из переменной число, чтобы получить результат.

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

Базовые операции:

$\otimes$	$\oplus$ in $\lambda^a\mathcal{M}^a$
$+$	+
$-$	−
$\times$	*
$/$	/
%	%
$<$	<
$>$	>
$\leq$	<=
$\geq$	>=
$=$	=
$\neq$	!=
$\wedge$	&&
$\vee$	!!

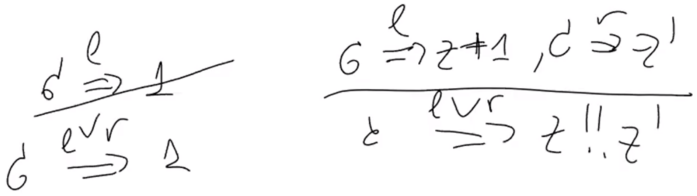
Пример программы: $1 \wedge (1/0)$. Пример описания:



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:



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

Обобщение правил $\Rightarrow_{\mathscr{E}}$ в семантику $[\![\bullet]\!]_{\mathscr{E}}$:

$$\frac{\sigma \overset{e}{\Longrightarrow}_{\mathscr{E}} n}{[\![e]\!]_{\mathscr{E}}\sigma = n}$$

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

$$\begin{array}{rcl} \mathscr{S} & = & \mathbf{skip} \\ & & \mathscr{X} := \mathscr{E} \\ & & \mathbf{read}\,(\mathscr{X}) \\ & & \mathbf{write}\,(\mathscr{E}) \\ & & \mathscr{S}\,;\mathscr{S} \end{array}$$

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:

$$[\![\bullet]\!] : \mathscr{S} \mapsto \mathbb{Z}^\star \to \mathbb{Z}^\star$$

Семантика большого шага связывает две конфигурации и программу:

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

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:

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

Мир — это инкапсуляция входного и выходного потока:

$$\mathscr{W} : \mathbb{Z}^\star \times \mathbb{Z}^\star$$

Определим операции:

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

Здесь $xi \in \mathbb{Z}^\star$ — 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.

$$c \xrightarrow{\texttt{skip}}_{\mathscr{S}} c \qquad [\textsc{Skip}]$$

$$\langle \sigma, \omega \rangle \xrightarrow{\texttt{x := e}}_{\mathscr{S}} \langle \sigma[x \leftarrow [\![e]\!]_{\mathscr{E}}\, \sigma], \omega \rangle \qquad [\textsc{Assign}]$$

$$\frac{\langle z, \omega' \rangle = \textbf{read}\ \omega}{\langle \sigma, \omega \rangle \xrightarrow{\texttt{read}\ (x)}_{\mathscr{S}} \langle \sigma[x \leftarrow z], \omega' \rangle} \qquad [\textsc{Read}]$$

$$\langle \sigma, \omega \rangle \xrightarrow{\texttt{write}\ (e)}_{\mathscr{S}} \langle \sigma, \textbf{write}\ ([\![e]\!]_{\mathscr{E}}\, \sigma)\, \omega \rangle \qquad [\textsc{Write}]$$

$$\frac{c_1 \xrightarrow{S_1}_{\mathscr{S}} c' \qquad c' \xrightarrow{S_2}_{\mathscr{S}} c_2}{c_1 \xrightarrow{S_1 ; S_2}_{\mathscr{S}} c_2} \qquad [\textsc{Seq}]$$

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:

$$[\![int]\!] : \mathscr{L} \to I \to O$$

$$[\![int]\!]_{\mathscr{L}}(p, x) = [\![p]\!]_{\mathscr{M}}(x)$$

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

Обобщим семантику оператора и определим $[\![\bullet]\!]_{\mathscr{S}}$:

$$\frac{\langle \Lambda, \langle i, \varepsilon \rangle \rangle \xrightarrow{S}_{\mathscr{S}} \langle \sigma, \omega \rangle}{[\![S]\!]_{\mathscr{S}} i = \textbf{out}\ \omega}$$

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)

$$
\begin{array}{rcl}
\mathscr{I} & = & \texttt{BINOP} \otimes \\
 & & \texttt{CONST}\ \mathbb{N} \\
 & & \texttt{READ} \\
 & & \texttt{WRITE} \\
 & & \texttt{LD}\ \mathscr{X} \\
 & & \texttt{ST}\ \mathscr{X} \\
\mathscr{P} & = & \varepsilon \\
 & & \mathscr{I}\ \mathscr{P}
\end{array}
$$

Программа стековой машины является списком (возможно пустым) инструкций.

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

$$\mathscr{C}_{SM} = \mathbb{Z}^\star \times \mathscr{C}$$

Правила семантики большого шага для СМ:

$$c \xrightarrow{\varepsilon}_{\mathscr{I}\mathscr{M}} c \qquad [\textsc{Stop}_{SM}]$$

$$\frac{\langle (x \oplus y)s, c\rangle \xrightarrow{p}_{\mathscr{I}\mathscr{M}} c'}{\langle yxs, c\rangle \xrightarrow{[\texttt{BINOP}\otimes]p}_{\mathscr{I}\mathscr{M}} c'} \qquad [\textsc{Binop}_{SM}]$$

$$\frac{\langle zs, c\rangle \xrightarrow{p}_{\mathscr{I}\mathscr{M}} c'}{\langle s, c\rangle \xrightarrow{[\texttt{CONST}\ z]p}_{\mathscr{I}\mathscr{M}} c'} \qquad [\textsc{Const}_{SM}]$$

$$\frac{\langle z, \omega'\rangle = \textbf{read}\ \omega,\ \langle zs, \langle \sigma, \omega'\rangle\rangle \xrightarrow{p}_{\mathscr{I}\mathscr{M}} c'}{\langle s, \langle \sigma, \omega\rangle\rangle \xrightarrow{\texttt{READ}\ p}_{\mathscr{I}\mathscr{M}} c'} \qquad [\textsc{Read}_{SM}]$$

$$\frac{\langle s, \langle \sigma, \textbf{write}\ z\, \omega\rangle\rangle \xrightarrow{p}_{\mathscr{I}\mathscr{M}} c'}{\langle zs, \langle \sigma, \omega\rangle\rangle \xrightarrow{\texttt{WRITE}\ p}_{\mathscr{I}\mathscr{M}} c'} \qquad [\textsc{Write}_{SM}]$$

$$\frac{\langle (\sigma\, x)s, \langle \sigma, \omega\rangle\rangle \xrightarrow{p}_{\mathscr{I}\mathscr{M}} c'}{\langle s, \langle \sigma, \omega\rangle\rangle \xrightarrow{[\texttt{LD}\ x]p}_{\mathscr{I}\mathscr{M}} c'} \qquad [\textsc{LD}_{SM}]$$

$$\frac{\langle s, \langle \sigma[x \leftarrow z], \omega\rangle\rangle \xrightarrow{p}_{\mathscr{I}\mathscr{M}} c'}{\langle zs, \langle \sigma, \omega\rangle\rangle \xrightarrow{[\texttt{ST}\ x]p}_{\mathscr{I}\mathscr{M}} c'} \qquad [\textsc{ST}_{SM}]$$

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:

$$\frac{\langle \varepsilon, \langle \Lambda, \langle i, \varepsilon\rangle\rangle\rangle \xrightarrow{p}_{\mathscr{I}\mathscr{M}} \langle s, \langle \sigma, \omega\rangle\rangle}{[\![p]\!]_{\mathscr{I}\mathscr{M}}\ i = \textbf{out}\ \omega}$$

Компилятор из языка операторов в стековую машину:

$$[\![\bullet]\!]^{comp} : \mathscr{S} \mapsto \mathscr{P}$$

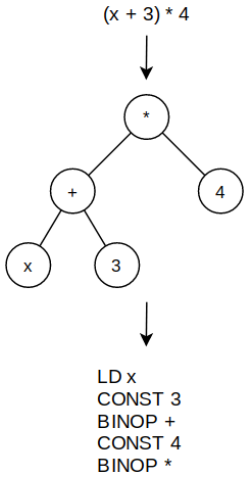
Компилятор — это функция, которая отображает программу на одном языке в программу на другом, сохраняя при этом семантику.

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

Опишим 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):

$$\begin{aligned}
[\![x]\!]^{comp}_{\mathscr{E}} &= [\texttt{LD}\ x] \\
[\![n]\!]^{comp}_{\mathscr{E}} &= [\texttt{CONST}\ n] \\
[\![A \otimes B]\!]^{comp}_{\mathscr{E}} &= [\![A]\!]^{comp}_{\mathscr{E}} [\![B]\!]^{comp}_{\mathscr{E}} [\texttt{BINOP}\ \otimes]
\end{aligned}$$

Пример компиляции:



Компилятор операторов:

$$\begin{aligned}
[\![x := e]\!]^{comp} &= [\![e]\!]^{comp}_{\mathscr{E}} [\texttt{ST}\ x] \\
[\![\textbf{read}\ (x)]\!]^{comp} &= [\texttt{READ}\,][\,\texttt{ST}\ x] \\
[\![\textbf{write}\ (e)]\!]^{comp} &= [\![e]\!]^{comp}_{\mathscr{E}} [\texttt{WRITE}] \\
[\![S_1; S_2]\!]^{comp} &= [\![S_1]\!]^{comp} [\![S_2]\!]^{comp}
\end{aligned}$$

$$[\![skip]\!]^{comp} = \epsilon$$

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 выполнять регрессионне тесты. Чтобы их пройти нужно

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

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 комментарии к пул реквесту указать кто я.

Установка компилятора языка Lama:

- 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/insta
    ll.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

Обновляем репу:

`Mov (opnd, opnd`	Копирует значение из первого во второй операнд
`Binop`	делает бинарную операцию
`IDiv`	Деление целого
`Cltd`	s

Разницы между X86 и стековой машиной практически нет. У обоих есть стек, мир и состояние. Здесь входным потоком будет stdin, выходным — stdout. Память будет состоянием.

Регистров в стековой машине нет. Но их нужно использовать при генерации кода. Будем использовать символическую интерпретацию. Будем генерировать код с помощью символического интепретатора для стековой машины. На стеке будут лежать не данные, а размещения (регистр, позиция на стеке, либо переменная в памяти).

Раньше компилятор получал конфигурацию и код. Сейчас компилятор будет получать окружение и код. Окружение можно считать объектом.

Размещения:

`R (id)`	регистр
`S (int)`	позиция на стеке
`M (string)`	именованная локация в памяти
`L (int)`	k

Регистры:

`ebx`	регистр (ebx, ecx, esl,
`ecx`	делает бинарную операцию
`esl`	Деление целого
`edl`	k
`eax`	Деление
`edx`	Деление
`ebp`	
`esp`	

READ – отводим новую позицию на символическом стеке (сначала выделяются регистры, затем стек). Получаем новую позицию и новое окружение. Возвращается пара — новое окружение и сгенерированный код

env.allocate – выделить новую позицию на стеке.

ST (x) снять с символического стека, засунуть в переменную