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:

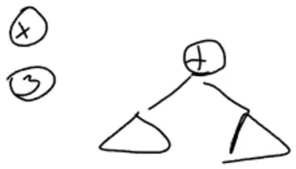
$$\begin{array}{rcl} \mathscr{X} & = & \{x, y, z, \ldots\} \\ \otimes & = & \{+, -, \times, /, \%, <, \leq, >, \geq, =, \neq, \vee, \wedge\} \\ \mathscr{E} & = & \mathscr{X} \\ & & \mathbb{N} \\ & & \mathscr{E} \otimes \mathscr{E} \end{array}$$

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:

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

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

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

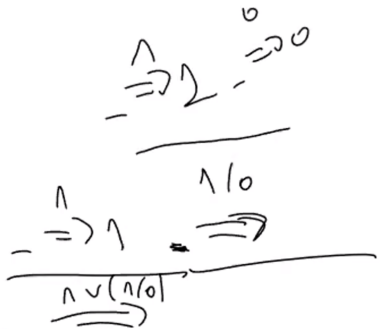
В случае с переменной мы должны извлечь из переменной число, чтобы получить результат.

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

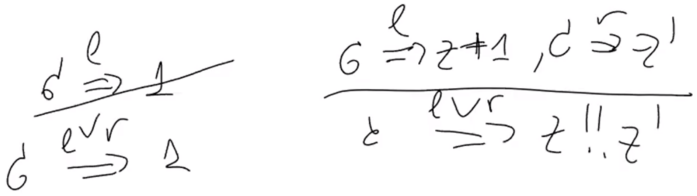
Пример программы: $1 \wedge (1/0)$. Пример описания:



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:

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

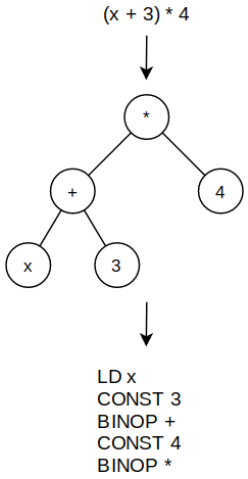
Компилятор — это функция, которая отображает программу на одном языке в программу на другом, сохраняя при этом семантику.

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
```

regression

2 Вторая домашка

Задача: реализовать компилятор CM в GAS x86.

Ветка для решения: `A02-straight-line-x86`. Нужно использовать код из предыдущей домашки.

```
git remote rename origin fork
git remote add original https://github.com/danyaberezun/compilers-2021-spring
git fetch original
git checkout -b A02-straight-line-x86 original/A02-straight-line-x86
```

`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

Важно создавать переменную во время чтения, если она не определена.

3 Третья домашка

Частично корректные компиляторы расширяют область определения программ. Они не строго соответсвуют семантике. Ее поведение может оличаться после компиляции. Эталонный интерпретатор полностью повторяет операционную семантику.

Компиляторы как правило является частично корректными. Просто потому что это сложно. Частичная коррекция достигается не специально.

Результат какой-то программы определяется ее семантикой или эталонным интерпретатором.

Тьюринг полнота — это возможность реализовать любую вычислимую функцию.

Вычислимая функция:

$$f : \mathbb{N}^k \to \mathbb{N}$$

На данный момент наш язык позволяет реализовать только полиномы, он не является тьюринг-полным.

Тут написать про оптимальность компилятора, перерисовать. Минимум шагов.

Задача оптимальной компиляции даже нашего языка неразрешимая.

10-я проблема Гильберта. Пусть есть многочлен с n переменными и целыми коэффициентами $p(x_1, x_2, \cdots, x_k)$. Корни ищатся перебором. Если к нулю не сходится, то будет выполняться бесконечно. Процедуры, которая позволит найти корни без перебора не существует.

Синтаксический анализ занимает меньшаю часть компилятора (четверь или десятую).

Синтаксически центричный принцип создания компилятора. Конкретный синтаксис языка программирования задается КС грамматикой. Конкретных синтаксисов у одного языка может быть несколько (например поддержка ключевых слов на русском языке).

Синтаксический анализатор преобразует конкретный синтаксис в абстрактный.

Сейчас у нас есть абстрактный синтаксис, но нет конкретного.

Дерево вывода КС грамматики является деревом абстрактного синтаксиса.

Проблемы:

На самом деле больниство языков (даже простых) синтаксис не является КС. Пример для языка C (конструкция `a * b`):

```
typedef int a;

a * b;

int f (int a, int b) {
    return a * b;
}
```

Во-вторых, дерево вывода не является АСД, так как один язык может иметь несколько грамматик, а АСД одно.

В-третьих, грамматика может быть контекстно зависима. Например,

```
<A <B>> f () {}

a >> b
```

Терминал » означает разное в зависимости от своей синтаксической позиции.

Отступы не значащие, но не строго. Напримерм, в си нельзя удалить пробелы здесь (это еще одно подствержение, что грамматика не КС):

```
void f () { ... }
```

В реальности используются монадические парсер-комбинаторы.

Атрибутные грамматики — это КС грамматика, где для каждого нетерминала указано 2 множества атрибутов. Атрибуты бывают наследуемые и синтезируемые. Если воспринимать нетерминал как процедуру, которая распознает что-то во входном потоке. Наследуемые атрибуты — это фаормальные параметры распознающей процедуры, а синтезируемые атрибуты — это результат.

То есть раньше КС-грамматика распознавала строку и возвращала 0 или 1 (результат распознавания), а сейчас грамматика параметризована и может возвращать больше результатов.

L-атрибутные грамматики.

L -> R1 R2 ... R3

Схема L-атрибутного анализа:

```
L[a] -> x=R1[a] y=R2[x] z=R3[y] {z}
```

Наледуемый атрибут можно использовать везе. Терминалы могут использовать синтезируемые атрибуты предыдущих терминалов. Результатом является z

Атрибуты позволяют выразить контестную зависимость.

Будем использовать левоатрибутный анализ.

Как реализовывать такие грамматики?

Есть монадические парсер-комбинаторы. Комбинатор — это функция высшего порядка, она комбинирует функции. Парсер принимает входной поток и что-то простое распознает в нем.

Фукнция высшего порядка — это функция, принимающая в качетсве аргументов другие функции или возвращающая другую функцию в качетсве результата.

```
fun (stream) -> Ok (result, stream) | fail (desc)
```

Получает список, возвращает результат и остаток списка.

Парсер комбинатор принимает что-то и возвращает парсер token принимает t и возвращает функцию, которая распознает это t.

atl — парсер альтернативы. Является левопредпочтительным, так как возвращает результат левого парсера, если распознавание с его помощью удалось. Иначе применяется правый парсер.

Монадические комбинаторы: seq.

Данный комбинатор комбинирует парсер и функцию, которая получает результат предыдущего парсера и возвращает парсер.

Ostap улучшенная реализация.

Лама содержит в себе DSL (предмето ориентированный язык) для описания синтаксического анализатора.

Начинается символом syntax. Является расширенной формулой бэкуса наура.

Lexer.lama — лексический анализатор. Набор примитивных парсеров. Содержит регулярки для терминалов (число, отступ)

Нужно дописать Parser.lama

memo — мемоизатор. eta — расширение, которое делает функцию из того, что следует далее.

Driver.lama — драйвер. Компонент компилятора, с которого начинается его работа. Разбирает опции командной строки, что-то запускает...

Пример простого парсера:

```
fun eof (stream) {
  case stream of
    {} -> Ok ({}, stream)
  | _  -> Fail ("EOF expected")
  esac
}
```

Комбинатор — это функция, которая получает действие и возвращает измененное действие.

Пример парсер-комбинатора, распознающего символ:

```
fun token (t) {
  fun (stream) {
    case stream of
      t0 : stream0 -> if compare (t0, t) == 0
                      then Ok (t, stream0)
                      else Fail (t.string ++ " expected")
                      fi
    | _            -> Fail (t.string ++ " expected")  поток пустой
    esac
  }
}
```

В данном случае действием является распознающая символ функция, которая определяется самим комбинатором и изменяется так, чтобы распознавать определенный символ.

Пример монадического парсер-комбинатора:

```
fun seq (p, q) {    p - парсер, q - комбинатор
  fun (stream) {
    case p (stream) of
      failure@Fail (_)     -> failure
    | Ok (result, stream0) -> q (result) (stream0)
    esac
```

```
  }
}
```

Монада определяет следующее вычисление на основе результата предыдущего.

Монадический парсер-комбинатор реализует схему L-атрибутной грамматики.

В модуле Ostap улучшенная реализация парсеров.

Семантические терминалы называются semantic action

4 Четвертая домашка

Написать оптимальный компилятор невозможно

Расширим язык операторов структурным потоком управления (structural control flow).

Называется структурным, так как нет перехода по метке (оператор goto).

$$\mathscr{S} \quad += \quad \texttt{if}\ \mathscr{E}\ \texttt{then}\ \mathscr{S}\ \texttt{else}\ \mathscr{S}$$

$$\texttt{while}\ \mathscr{E}\ \texttt{do}\ \mathscr{S}$$

Добавляем два оператора. Они записаны в абстрактном синтаксисе, но в символьной форме. В виде дерева их можно записать так:

В конкретном синтаксисе обязательно присутсвуют закрывающие конструкции (fi, od).

Операционная семантика для новых конструкций состоит из 4 правил:

$$\frac{\sigma \xRightarrow{e}_{\mathscr{E}} n \neq 0 \qquad \langle \sigma, w\rangle \xRightarrow{S_1}_{\mathscr{S}} c'}{\langle \sigma, w\rangle \xRightarrow{\texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2}_{\mathscr{S}} c'} \quad [\textsc{If-True}]$$

$$\frac{\sigma \xRightarrow{e}_{\mathscr{E}} 0 \qquad \langle \sigma, w\rangle \xRightarrow{S_2}_{\mathscr{S}} c'}{\langle \sigma, w\rangle \xRightarrow{\texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2}_{\mathscr{S}} c'} \quad [\textsc{If-False}]$$

$$\frac{\sigma \xRightarrow{e}_{\mathscr{E}} n \neq 0 \qquad \langle \sigma, w\rangle \xRightarrow{S}_{\mathscr{S}} c' \qquad c' \xRightarrow{\texttt{while}\ e\ \texttt{do}\ S}_{\mathscr{S}} c''}{\langle \sigma, w\rangle \xRightarrow{\texttt{while}\ e\ \texttt{do}\ S}_{\mathscr{S}} c''} \quad [\textsc{While-True}]$$

$$\frac{\sigma \xRightarrow{e}_{\mathscr{E}} 0}{\langle \sigma, w\rangle \xRightarrow{\texttt{while}\ e\ \texttt{do}\ S}_{\mathscr{S}} \langle \sigma, w\rangle} \quad [\textsc{While-False}]$$

Семантика есть описание поведения программы.

Добавим синтаксическое расширение. Или синтаксичекий сахар. Это расширение которое существует в конкретном синтаксисе, но не в абстрактном, в абстрактный они проецируются.

```
   if e1 then s1  ->  if e1 then s1
 elif e2 then s2      else if e2 then s2
 else s3                   else sk
```

```
  fi                          fi
                        fi
```

Достаточно добавить изменения в синтаксический анализатор (парсер), чтобы все заработало.

```
for s1, e, s2 do s3 od -> s1;
                          while e do
                           s3; s2
                          od
```

Цикл с постусловием:

```
do s while e od   ->   s;
                       while e do
                         s
                       od
```

Реализовывать его при помощи синтаксического расширения ужасно, так как получаем экспоненциальную сложность в преобразовании

```
    do                do
      do              while od
      while od   ->   while do      -> ...
    while od            do
                        while od
                      od
```

Если при синтаксическом расширении получается нелинейное преобразование, то лучше его не реализовывать. Поэтому будем добавлять эту конструкцию на уровне абстрактного синтаксиса. Дз определить семантику.

5 Пятая домашка

Задача: расширить компиляцию в стековую машину и X86.

На данный момент язык стековой машины не достаточно выразителен, чтобы отобразить циклы. Поэтому расширим ее, то есть добавим новые конструкции и опишим семантику для них. множество меток:

$$\mathscr{L} = \{l_1, l_2, \cdots\}$$

$$\begin{aligned} \mathscr{I} \quad += \quad & \texttt{LABEL}\ \mathscr{L} \\ & \texttt{JMP}\ \mathscr{L} \\ & \texttt{CJMP}_x\ \mathscr{L},\ \text{where } x \in \{\texttt{nz}, \texttt{z}\} \end{aligned}$$

JMP — инструкция безусловного перехода к метке, CJMP — инструкция перехода по 0 или 1 (значение берется из стека).

В описании семантики будем использовать окружение (environment) Γ. В нашем случае окружением является программа P (последоваательность инструкций) ,

которую мы интерпретируем. $P[l]$ — подпрограмма P которая начинается с метки l. В последовательсноти инструкций ..., LABEL l ... правая часть и есть $P[l]$.

В семантических правилах будем исходную программу обозначать P, а p — хвост, до которого мы добрались.д

$$\frac{p \vdash \langle \varepsilon, \langle \Lambda, \langle i, \varepsilon \rangle \rangle \rangle \xrightarrow{p}_{\mathscr{SM}} \langle s, \langle \sigma, \omega \rangle \rangle}{[\![p]\!]_{\mathscr{SM}}\, i = \mathbf{out}\ \omega}$$

$$\frac{P \vdash c \xrightarrow{p}_{\mathscr{SM}} c'}{P \vdash c \xrightarrow{[\text{LABEL } l]p}_{\mathscr{SM}} c'} \quad [\text{LABEL}_{SM}]$$

$$\frac{P \vdash c \xrightarrow{P[l]}_{\mathscr{SM}} c'}{P \vdash c \xrightarrow{[\text{JMP } l]p}_{\mathscr{SM}} c'} \quad [\text{JMP}_{SM}]$$

$$\frac{z \neq 0, \quad P \vdash \langle s, \theta \rangle \xrightarrow{P[l]}_{\mathscr{SM}} c'}{P \vdash \langle zs, \theta \rangle \xrightarrow{[\text{CJMP}_{nz}\ l]p}_{\mathscr{SM}} c'} \quad [\text{CJMP}^{+}_{nz\,SM}]$$

$$\frac{z = 0, \quad P \vdash \langle s, \theta \rangle \xrightarrow{p}_{\mathscr{SM}} c'}{P \vdash \langle zs, \theta \rangle \xrightarrow{[\text{CJMP}_{nz}\ l]p}_{\mathscr{SM}} c'} \quad [\text{CJMP}^{-}_{nz\,SM}]$$

$$\frac{z = 0, \quad P \vdash \langle s, \theta \rangle \xrightarrow{P[l]}_{\mathscr{SM}} c'}{P \vdash \langle zs, \theta \rangle \xrightarrow{[\text{CJMP}_{z}\ l]p}_{\mathscr{SM}} c'} \quad [\text{CJMP}^{+}_{z\,SM}]$$

$$\frac{z \neq 0, \quad P \vdash \langle s, \theta \rangle \xrightarrow{p}_{\mathscr{SM}} c'}{P \vdash \langle zs, \theta \rangle \xrightarrow{[\text{CJMP}_{z}\ l]p}_{\mathscr{SM}} c'} \quad [\text{CJMP}^{-}_{z\,SM}]$$

Метка ничего не делает, из прошлой конфигурации выполняем хвост. Безусловный переход изменяет только поток управления.

$[\![c]\!] - - - SMc., .CJMP_z l_s 2$

$$\begin{array}{l} \textbf{if } c \\ \textbf{then } s_1 \\ \textbf{else } s_2 \\ \textbf{fi} \end{array} \quad \rightarrow \quad \begin{array}{l} [\![c]\!] \text{ — код SM для } c \\ \textbf{CJMP}^{+}_{z}\ l_{s_2} \\ [\![s_1]\!] \\ \textbf{JMP } l_{\text{eof}} \\ \textbf{LABEL } l_{s_2} \\ [\![s_2]\!] \\ \textbf{LABEL } l_{\text{eof}} \end{array} \tag{1}$$

Проблема в такой генерации возникает, когда встречается множество вложенных условий. Тогда переход для выхода будет совершаться многократно (линейно).

Процессоры общего назначения (ARM, X86) устроены так, что используют предсказание переходов. Условые переходы выполняются заранее, поэтому имеют нулевую вычислительную стоимость (процессор заранее берет инструкции для выполненеия из нужного места). Для условных переходов выполняется угадывание,

которое почти всегда верно, но стоимость ошибки выше, чем переход.

$$\begin{array}{l} \textbf{while}\, c \\ \textbf{do}\, s \\ \textbf{od} \end{array} \quad \rightarrow \quad \begin{array}{ll} \textbf{LABEL}\ l_c & \textbf{JMP}\ l_c \\ [\![c]\!] & \textbf{LABEL}\ l_s \\ \textbf{CJMP}_z^+\ l_{\text{eof}} & [\![s]\!] \\ [\![s]\!] & \textbf{LABEL}\ l_c \\ \textbf{JMP}\ l_c & [\![c]\!] \\ \textbf{LABEL}\ l_{\text{eof}} & \textbf{CJMP}_{nz}^+\ l_s \end{array} \tag{2}$$

Вторая генерация более правильная, так как в теле циклав выполняется только одна команда перехода, хотя такая разница незначительная для современных процессоров.

Хвост программы по метке сохраняется в окружении.

Метка переходит в метку. бЕзусловный переход тв безусловный. Условный переход испльзует СМ использует стек, в x86 нужно смотреть в символический стек. Значение с символического стека нужно переместить в регистр флагов, так как инструкция условного перехода использует их (cmp сравнивать с нулем).

Рассинхронизация символического стека не происходит, так как в момент перехода имеем пустой стек.