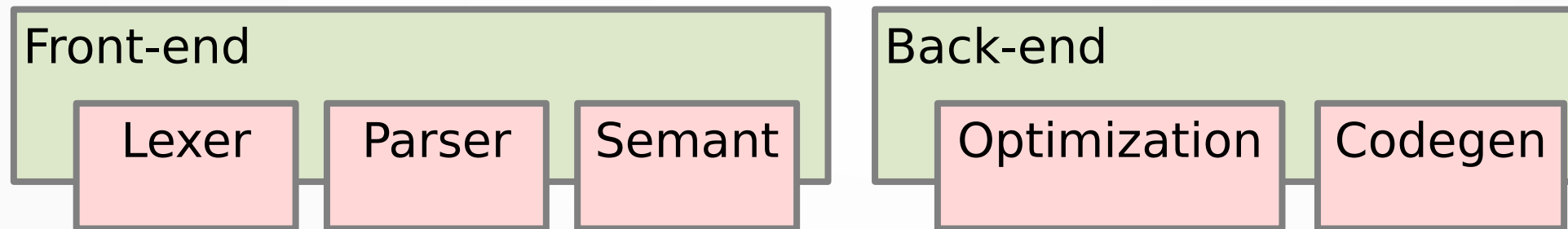


Lab 3. Part I. Semantic analyzer
for cool.

Mipt (Ilab), 28.11.2018

Semantic analyzer intro



Проверки, в семантическом анализе:

- Все идентификаторы определены
- **Соответствие типов**
- Отношения наследования
- Определения не повторяются (определения классов, методов)
- Не используются зарезервированные слова

Область видимости (scope).

Под областью видимости – понимаем часть программы, где есть доступ к идентификатору.

Большинство языков имеют статическую область видимости – scope определяется только текстом программы не может быть переопределен в run-time.

Динамическая область видимости означает, что область видимости переменной зависит от того, как именно исполнится программа. Языки с динамической областью видимости: Lisp, Bash.

Ex 1:

```
let y:String <- «abc» in y + 3
```

Type error

Ex 2:

```
let y: Int in x + 3
```

x defined ?

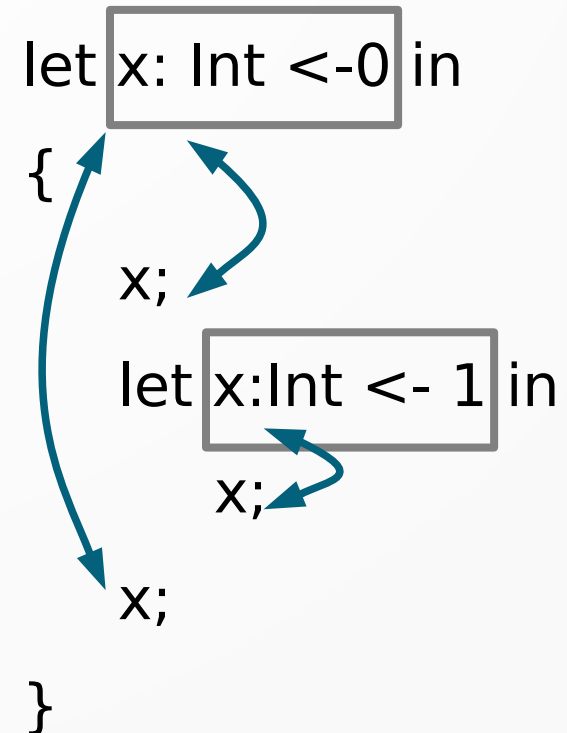
Scope identifiers in Cool

Области видимости в cool:

- let expressions
- class declarations
- attribute definitions
- method definitions
- formal parameters
- case expressions

Let expressions


Example :



Class, attributes and methods.

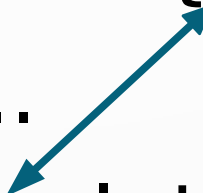
Class name are in global scope:

```
Class Foo {  
    ... let y : Bar in ...  
};  
class Bar {  
    ...  
};
```



Attribute names are global in class:

```
Class Foo {  
f(): Int {a};  
    ...  
a : Int <- 0;  
};
```



It's OK.

Методы:

Могут определяться в родительском классе

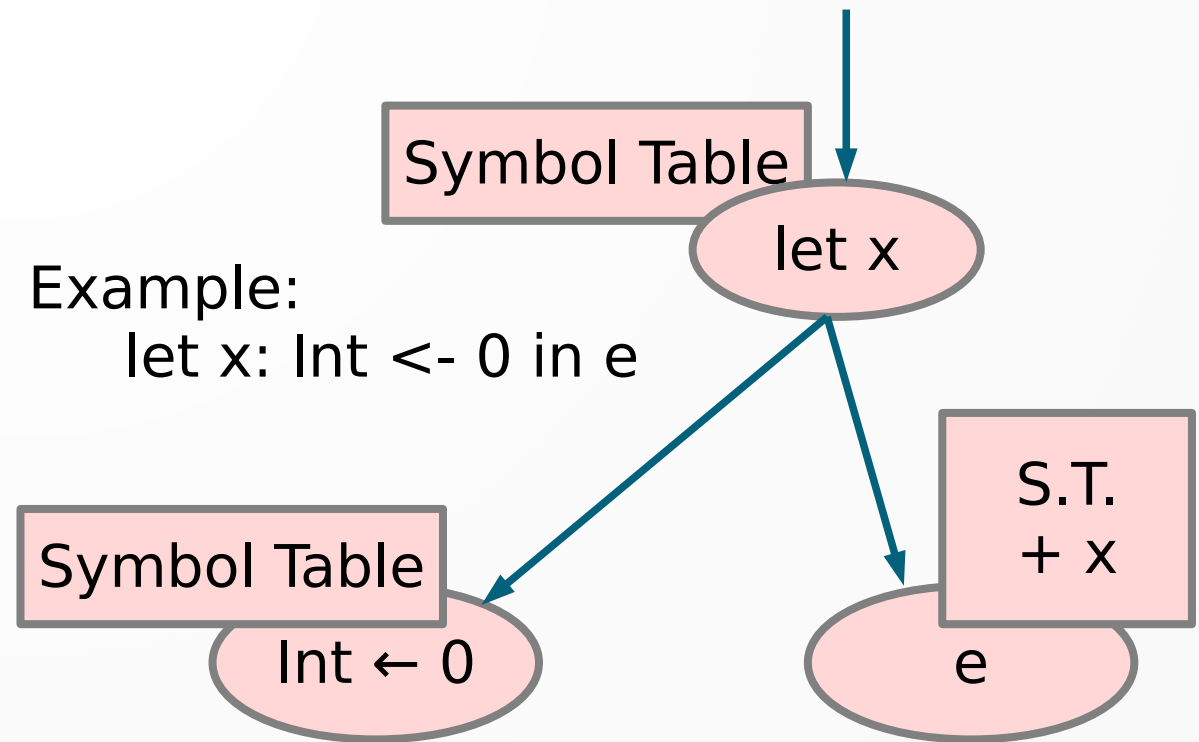
Могут определяться в родительском классе

Могут переопределять родительский метод

Таблица символов. Обход.

Обход дерева,
построенного на этапе
синтаксического анализа,
при выполнении
семантического анализа
можно разбить на:

- Before: обработка AST ноды n
- Recurse: обработка одного из детей ноды
- After: завершение обработки AST ноды n



то есть x определяется до
выполнения поддерева e и
удаляется после

Таблица символов. Идея.

Before - добавить определение x к списку текущих определений и перекрыть любые другие определения x

`let x: Int <- 0 in e`

Recurse - вызвать

After - обработать тело `let (e)`, удалить определение x и восстановить старые определения x (до вызова `let`)

=> Необходимы операции с таблицами:

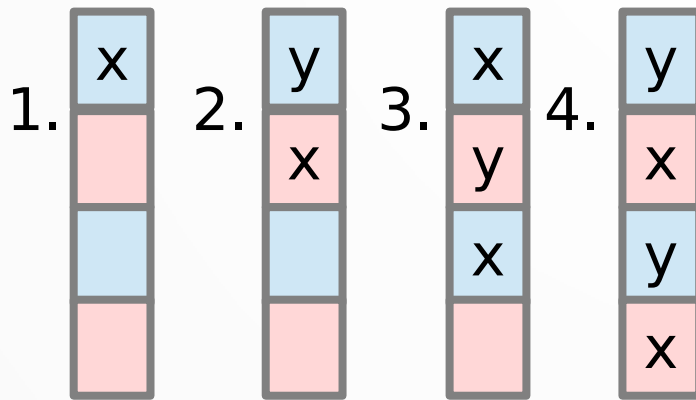
- `add_symbol(x)` - добавить символ **в стек**
- `find_symbol(x)` - найти первое вхождение символа
- `remove_symbol()` - удалить символ (**pop from stack**)

При выходе
из scope



Таблица символов. Пример.

Ex:



При добавлении нового элемента в локальной области видимости, элемент попадает на вершину стека и будет найден первым.

Таблица символов. Ошибки.

Чтобы работать со стеком было удобно - необходимо расширить функциональность работы с таблицами (add, find, remove):

- `enter_scope()` - начинает новую область видимости
- `check_scope(x)` - проверяет, объявлен ли `x` в текущей области видимости
- `exit_scope()` - очищает текущую область видимости

Если использовать `exit_scope`, то можно обойтись без `remove_symbol`.

Таблица символов. Резюме.

Вернемся к слайду 5.

Легко видеть, что...

Не получится проверить имена классов за один проход.

Таблица символов – структура данных, которая отслеживает идентификаторы в конкретной точке AST.

Проверка типов. Первый уровень.

Определим понятие тип как множество значений и множество операций над ними (понятие класс вполне подходит под такое описание).

Ex:

`add %r1, %r2, %r3`

Берет данные из %r2 и %r3,
складывает их и записывает в %r1

Вопрос - какого типа %r1, %r2 и %r3

Целое

С плавающей запятой

Указатель на данные

Указатель на функцию

Проверка типов. Первый уровень.

Вопрос для типов регистров - некорректен, правильный вопрос - сочетание каких типов имеет смысл?

Цель проверки типов - убедиться, что операции над ними корректны.

Важно: Никто кроме семантического анализатора не будет проверять корректность передаваемых значений.

Возможно три типа типизации:

- Статическая (C, Java, Cool)
- Динамическая (Perl, Scheme, Ruby, Lisp, Python)
- Отсутствие типов - (машинный код)

Проверка типов. Второй уровень.

Для формализации описания проверки типов в COOL используются правила (основанные на логике высказываний):

$$\begin{array}{l} O, M, C \vdash e_0 : T_0 \\ O, M, C \vdash e_1 : T_1 \\ \vdots \\ O, M, C \vdash e_n : T_n \\ T'_0 = \begin{cases} C & \text{if } T_0 = \text{SELF_TYPE}_C \\ T_0 & \text{otherwise} \end{cases} \\ M(T'_0, f) = (T'_1, \dots, T'_n, T'_{n+1}) \\ T_i \leq T'_i \quad 1 \leq i \leq n \\ T_{n+1} = \begin{cases} T_0 & \text{if } T'_{n+1} = \text{SELF_TYPE} \\ T'_{n+1} & \text{otherwise} \end{cases} \\ \hline O, M, C \vdash e_0.f(e_1, \dots, e_n) : T_{n+1} \end{array}$$

[Dispatch]



Проверка типов. Логика высказываний.

Правила логического вывода:

Если Гипотеза верна \Rightarrow то Заключение верно
if Hypothesis is true \Rightarrow then Conclusion is true

С точки зрения проверки типов:

if e_1 and e_2 have certain(определенный) types,
then e_3 has a certain type

Ex:

$$\left[(e_1 : Int \wedge e_2 : Int) \Rightarrow e_1 + e_2 : Int \right]$$

Проверка типов. Выводы.

Для того чтобы не писать каждый раз вывод (\Rightarrow):

$$Hyp_1 \wedge Hyp_2 \wedge \dots \wedge Hyp_n \Rightarrow Concl$$

используется ступенчатая запись:

$$\frac{\vdash Hyp_1 \vdash Hyp_2 \vdash \dots \vdash Hyp_n}{\vdash Concl}$$

Для того чтобы написать следующее выражение: «Получаем, что e имеет тип T » необходимо написать:

$$\vdash e : T$$

Проверка типов. Примеры.

$$\frac{i \text{ is an integer literal}}{\vdash i : Int} [Int] \qquad \frac{\vdash e_1 : Int \vdash e_2 : Int}{\vdash (e_1 + e_2) : Int} [Add]$$

Пример вывода типа:

**Вывод имеет
вид AST
деревя**

$$\frac{\frac{1 \text{ is an integer literal}}{\vdash 1 : Int} \quad \frac{2 \text{ is an integer literal}}{\vdash 2 : Int}}{\vdash 1 + 2 : Int}$$

Проверка типов. Связь с AST.

Одно правило используется для каждой ноды AST.
Типы рассматриваются снизу вверх.

$$\frac{}{\vdash \textit{false} : \textit{Bool}} \quad [\textit{False}]$$

$$\frac{s \text{ is an string literal}}{\vdash s : \textit{String}} \quad [\textit{String}]$$

$$\frac{}{\vdash \textit{new} T : T} \quad [\textit{New}]$$

$$\frac{\begin{array}{c} \vdash e_1 : \textit{Bool} \\ \vdash e_2 : T \end{array}}{\vdash \textit{while } e_1 \textit{ loop } e_2 \textit{ pool} : \textit{Object}} \quad [\textit{Loop}]$$

Проверка типов. Окружение.

$$\frac{x \text{ is a Variable}}{\vdash x : ?} [Var]$$

Для этого правила *недостаточно информации*,
необходима информация *из окружения*.

Переменная называется свободной, если определяется
вне выражения.

Если переменная не свободна, то она называется
связанной.

Type environment (окружение типа) дает информацию для
свободных переменных.

Проверка типов. Окружение.

Для свободных переменных необходима дополнительная информация:

O – ObjectIdentifiers

$O \vdash e : T$

Означает: если тип задан в O , то выражение e имеет тип T .
 O можно рассматривать как интерфейс к таблице символов.

Т.е. O ищет тип первого идентификатора с таким же именем.

$$\frac{O(x) = T}{\vdash x : T} \quad [Var]$$

Проверка типов. Окружение + область видимости.

Проблема : для let-выражений в таблице нет записей о типах переменных.

Решение:

$$\frac{O[T_0 / x] \vdash e_1 : T_1}{O \vdash \text{let } x : T_0 \text{ in } e_1 : T_1} \quad [Let - no - Init]$$

Здесь $O[T_0 / x]$ просто означает, что мы добавляем в O пару $x : T_0$

Более формальная запись:

$$O[T / c](c) = T$$

$$O[T / c](d) = O(d) \text{ if } d \neq c$$

Проверка типов. Let с инициализацией

$$\frac{\begin{array}{c} O \vdash e_0 : T_0 \\ O[T_0 / x] \vdash e_1 : T_1 \end{array}}{O \vdash \text{let } x : T_0 \leftarrow e_0 \text{ in } e_1 : T_1} \quad [\textit{Let} - \textit{Init}]$$

Проверка типов. Преобразования типов.

Проблема: В случаях, когда классы наследуются друг от друга семантический анализатор должен считать корректным преобразование из дочерних классов в родительские.

Решение: Конечно же усложнение описания!

Отношение подтипов в классах:

$X \leq Y := X \text{ inherits from } Y \text{ (or } X \text{ equal } Y)$

$$\frac{\begin{array}{c} O \vdash e_0 : T \\ O[T_0 / x] \vdash e_1 : T_1 \\ T_0 \leq T \end{array}}{O \vdash \text{let } x : T_0 \leftarrow e_0 \text{ in } e_1 : T_1} \quad [\text{Let} - \text{Init}]$$



Проверка типов. Атрибуты класса.

Просто нужно еще определение для всех атрибутов $x:T$ из конкретного класса C :

$$\frac{\begin{array}{c} O_c(x) = T_0 \\ O_c \vdash e_1 : T_1 \\ T_1 \leq T_0 \end{array}}{O_c \vdash x : T_0 \leftarrow e_1;} \quad [Attr - Init]$$

Проверка типов. Преобразование. Верхняя граница для наследования.

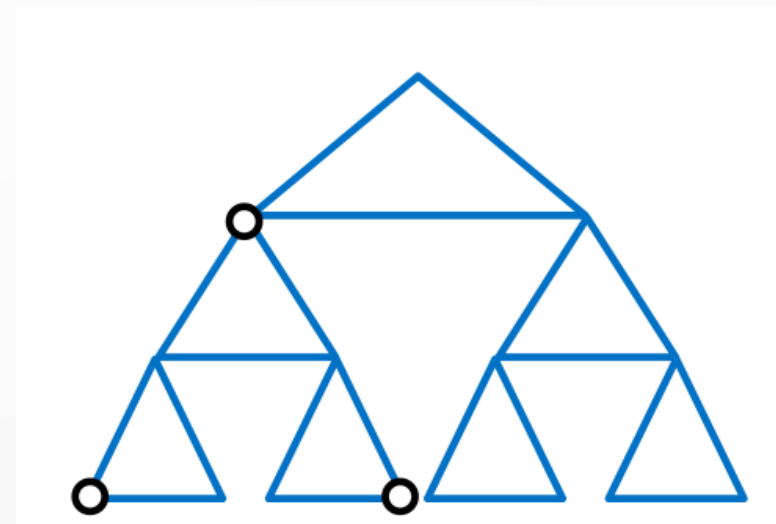
Для выражения *if e_0 then e_1 else e_2 fi* e_1 и e_2 не обязаны быть одного типа, достаточно чтобы результирующий тип был родительским для выведенных типов:

Тип должен быть самым близким к обоим выведенным типам T_1 и T_2 общим родителем.

Такой тип называется
least upper bound (lup):

least upper bound for X and Y

$$\text{lub}(X, Y) \Leftrightarrow \sqcup(X, Y)$$



Проверка типов. Преобразование.

$$\frac{\begin{array}{c} O \vdash e_0 : \textit{Bool} \\ O \vdash e_1 : T_1 \\ O \vdash e_2 : T_2 \end{array}}{O \vdash \textit{if } e_0 \textit{ then } e_1 \textit{ else } e_2 \textit{ fi} : \text{lub}(T_1, T_2)} \quad [\textit{If} - \textit{Then} - \textit{Else}]$$

Проверка типов. Верхняя граница для наследования. Пример.

```
Class Object
Class Bool inherits Object
Class Point inherits Object
Class Line inherits Object
Class Shape inherits Object
Class Quad inherits Shape
Class Circle inherits Shape
Class Rect inherits Quad
Class Square inherits Rect
```

```
lub(Point, Quad) = Object
lub(Square, Rect) = Quad
lub(Square, Rect) = Rect
lub(Square, Circle) = Object
?
```

Проверка типов. Методы.

Конечно у них такой же приоритет, как и у идентификаторов объектов.

Соответственно для них вводится кодировка как и для объектов:

$$\frac{\begin{array}{c} O \vdash e_0 : T_0 \\ O \vdash e_1 : T_1 \\ \dots \\ O \vdash e_n : T_n \end{array}}{O \vdash e_0.f(e_1, \dots, e_n) : ?} \quad [Dispatch]$$

$$\begin{array}{l} M(C, f) = (T_1, \dots, T_n, T_{n+1}) \\ f(x_1 : T_1, \dots, x_n : T_n) : T_{n+1} \end{array}$$

Проверка типов. Методы.

$$O, M \vdash e_0 : T_0$$

$$O, M \vdash e_1 : T_1$$

...

$$O, M \vdash e_n : T_n$$

$$M(T_0, f) = (T_{1'}, \dots, T_{n'}, T_{n+1})$$

$$T_i \leq T_{i'} \text{ for } 1 \leq i \leq n$$

$$\frac{}{O, M \vdash e_0.f(e_1, \dots, e_n) : T_{n+1}} \quad [Dispatch]$$

Проверка типов. Add

$$\frac{O, M, C \vdash e_1 : Int \quad O, M, C \vdash e_2 : Int}{O, M, C \vdash e_1 + e_2 : Int} [Add]$$

```
TypeCheck(Environment, e1 + e2) = {  
    T1 = TypeCheck(Environment, e1);  
    T2 = TypeCheck(Environment, e2);  
    Check T1 == T2 == Int;  
    return Int;  
}
```

Проверка типов. Let

$$\frac{\begin{array}{c} O, M, C \vdash e_0 : T_0 \\ O[T / x], M, C \vdash e_1 : T_1 \\ T_0 \leq T \end{array}}{O, M, C \vdash \text{let } x:T \leftarrow e_0 \text{ in } e_1 : T_1} \quad [\textit{Let} - \textit{Init}]$$

```
TypeCheck(Environment, let x:T <- e0 in e1) = {  
  T0 = TypeCheck(Environment, e0)  
  T1 = TypeCheck(Environment.add(x:T), e1)  
  Check subtype(T0, T1);  
  return T1;  
}
```

Краткий план работ

- В программе надо будет
 - Обойти все классы и построить графнаследования.
 - Убедится, что граф корректен.
 - Для каждого класса
 - (a) Обойти AST, добавляя все видимыеобъявления в таблице символов.
 - (b) Проверить каждое выражение на корректность типа.
 - (c) Заполнить AST типами.
- https://gitlab.com/mipt.igor.gorban/pa3_semant

Файлы

- **cool-tree.h** Этот файл содержит пользовательские расширения для узлов абстрактного синтаксического дерева. Вероятно, вам придется добавлять дополнительные объявления, но не изменять существующие объявления.
- **semant.cc** Это основной файл для реализации семантического анализа. Он содержит некоторые предопределенные символы для вашего удобства и начало реализации класса `ClassTable` для представления графа наследования. Вы можете использовать или игнорировать их. Семантический анализ начинается с вызова метода `semant()`. Объявление класса `program_class` находится в `cool-tree.h`. Любые методы, которые вы добавляете в `cool-tree.h`, должны быть реализованы в этом файле.
- **semant.h** Этот файл является файлом заголовка для `semant.cc`.
- **good.cl** и **bad.cl** Эти файлы проверяют несколько семантических функций.
- **README** Этот файл содержит подробные инструкции, а также ряд полезных советов.