***28. Промежуточное представление и оптимизации высокого и низкого уровня.***

***!!!Убери отсюда лишнее!!!  
Ничего не убрал, только еще добавил***

[*несколько слайдов отсюда можно почитать*](http://sp.cmc.msu.ru/proseminar/2011/belevantsev.2011.04.01.pdf) *– из предыдущего билета*

*Промежуточное представление(*[*простой пример*](http://citforum.ru/programming/theory/serebryakov/8.shtml)*):*

В процессе трансляции компилятор часто используют промежуточное представление (ПП) исходной программы, предназначенное прежде всего для удобства генерации кода и/или проведения различных оптимизаций. Сама форма ПП зависит от целей его использования.

[низкоуровневая и высокоуровневая оптимизация](https://www.google.ru/url?sa=t&rct=j&q=&esrc=s&source=web&cd=3&ved=0ahUKEwiJ1fexhu3QAhXDCiwKHeOrDDsQFggkMAI&url=http%3A%2F%2Fcompiler.itlab.unn.ru%2Fuploads%2Fcompiler_practice.update19.doc&usg=AFQjCNGiYWU26hpvpzgx81bxOdPIE5VI5g&sig2=ejCuTmWfseaNqsBLiWpjqQ&bvm=bv.141320020,d.bGg) – тут есть почти все пункты и про SSA (хорошо бы сюда всё нужное скопировать, но лень)

*Свойства промежуточного представления высокого уровня*

* Сохранение семантики исходного кода
  + Контексты блоков и функций
  + Типы данных
  + Общие подвыражения
  + Циклы в явном виде
* Сохранение метаданных
* Упрощение оптимизации
* Наилучший известный HIR - Граф управления (control flow graph)
  + Узлы – блоки, участки кода без переходов
  + Дуги – переходы между блоками
  + Возможно параллельное исполнение блоков
  + Хранение дополнительных данных в узлах и дугах
  + Для описания блоков используется LIR

Предложения программы организуются в *базовые блоки* (не обязательно максимальные). Базовым блоком может быть любой фрагмент кода не содержащий переходов. Программа входит в первое предложение такого блока и выходит из последнего.

*Граф управления (control flow graph) –* это направленный граф, в вершинах которого расположены базовые блоки, а дуги соответствуют переходам.

*Анализ потока управления/данных -* проход делит программу на базовые блоки. В процессе деления удаляются недостижимые циклы. Затем вычисляется время жизни каждого псевдорегистра. Также происходит обнаружение выражений, значения которых далее нигде не используются. Объединяются ссылки на ячейки памяти с инструкциями сложения или вычитания для дальнейшего их преобразования в автоинкрементную или автодекрементную адресацию.

Возможные улучшения: сокращение размера базовых блоков*.*

* Объединение графов управления отдельных функций
* Построение SSA Form
  + SSA Form (Single Static Assignment) – каждая переменная присваивается ровно один раз
* Выполнение Escape analysis
  + Escape analysis – нахождение максимального замкнутого подграфа управления содержащего все доступы к значению

*SSA:*

*Static Single Assignment Form (SSA Form)* – форма представления программы, в которой любой переменной значение присваивается не более одного раза.

SSA форма и граф управления потоком были предложены для представления потока данных и потока управления в программе. Каждая из этих ранее независимых технологий использовалась в классе оптимизаций. Большое число современных алгоритмов оптимизации программ основаны на совместном использовании графа управления и SSA-формы.

SSA – это форма программы в которой значение каждой переменной присваивается только один раз, но может читаться сколько угодно раз.(см. рисунки)





Каждое предложение внутреннего представления вычисляет некоторое выражение и использует результат для присваивания или перехода.

Преобразование программы в SSA форму – двухэтапный процесс:

1. На первом этапе добавляются тривиальные *Ф*-функции в некоторые вершины графа управления.
2. На втором этапе генерируются новые переменные (находятся зависимости, переменные получают «версии»).

*Массивы в SSA:*

Все обращения к элементам массивом оборачиваются специальными функциями.

Исключение составляет тот случай, если язык поддерживает операции с массивами как со скалярами (поэлементное копирование и пр.). В этом случае переменная массива обрабатывается как обычный скаляр.

Рассмотрим случай обращения к элементу массива.

Исходный код, использующий массивы:

<- A(i)

A(j) <- A(i)

<- A(k) + 2

Эквивалентный код, в котором использованы специальные операторы доступа:

<- Access(A, i)

A(j) <- Update(A, j, V)

T <- Access(A, k)

<- T + 2

SSA форма:

<- Access(A8, i7)

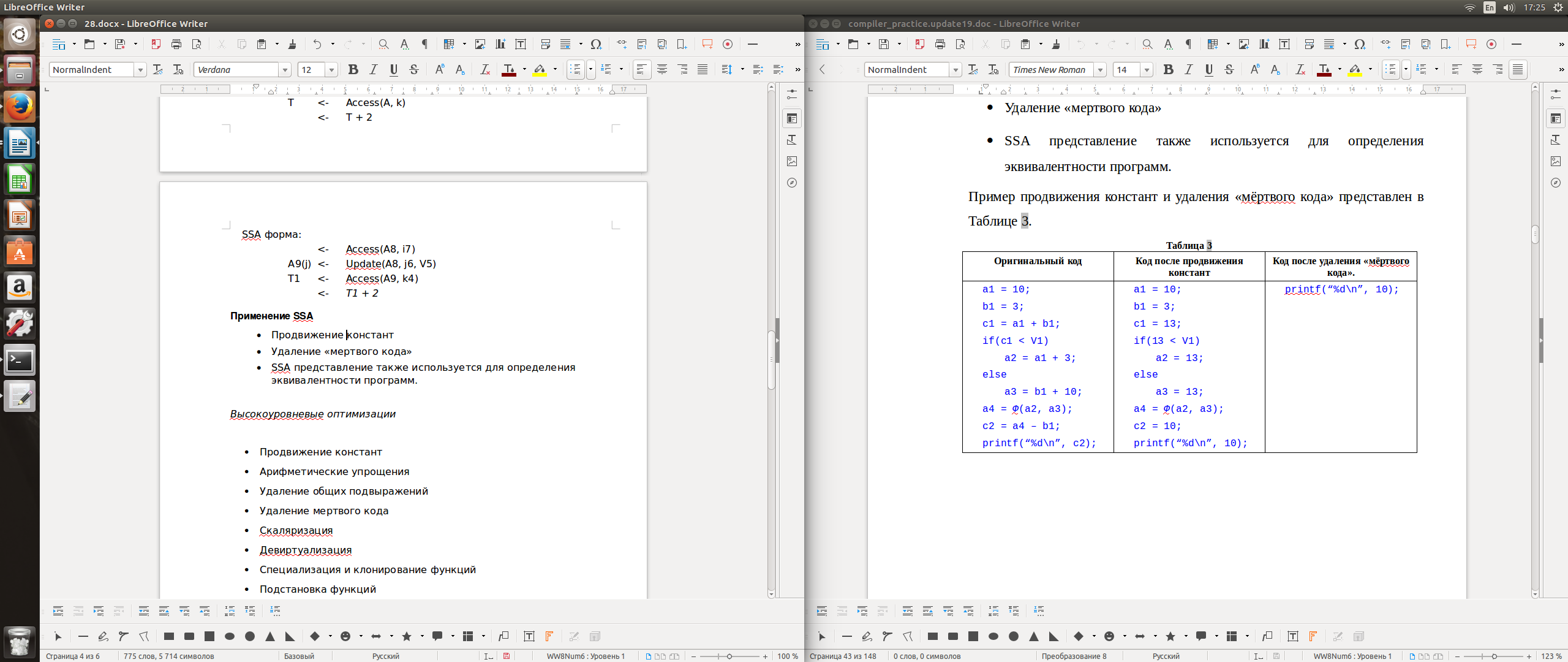
A9(j) <- Update(A8, j6, V5)

T1 <- Access(A9, k4)

*<- T1 + 2*

### Применение SSA

* Продвижение констант
* Удаление «мертвого кода»
* SSA представление также используется для определения эквивалентности программ.



*Высокоуровневые оптимизации*

*См* [*низкоуровневая и высокоуровневая оптимизация*](https://www.google.ru/url?sa=t&rct=j&q=&esrc=s&source=web&cd=3&ved=0ahUKEwiJ1fexhu3QAhXDCiwKHeOrDDsQFggkMAI&url=http%3A%2F%2Fcompiler.itlab.unn.ru%2Fuploads%2Fcompiler_practice.update19.doc&usg=AFQjCNGiYWU26hpvpzgx81bxOdPIE5VI5g&sig2=ejCuTmWfseaNqsBLiWpjqQ&bvm=bv.141320020,d.bGg) *страница 100*

* Продвижение констант
* Арифметические упрощения
* Удаление общих подвыражений
* Удаление мертвого кода
* Скаляризация
* Девиртуализация
* Специализация и клонирование функций
* Подстановка функций
* Векторизация циклов
* Удаление хвостовой рекурсии
* Вынесение инвариантного кода из циклов
* Распараллеливание блоков

*Свойства промежуточного представления низкого уровня*

* Абстракция системы команд процессора
  + Гибкое представление инструкций процессора
  + Расширяемость
* Удобство оптимизации
* Сохранение информации высокого уровня
* Стандартный способ – трехадресный код
  + трехадресный код, квад (quad, three address code (TAC)) – каждая инструкция имеет вид x:=у op z
  + Таким образом инструкция – четверка (op, arg0, arg1, result)
  + Не все инструкции задействуют все поля, например (goto, label, -, -)
  + Дополнительные данные хранятся в метках

*Низкоуровневые оптимизации*

* Замена цепочек суперинструкциями
* Конвейеризация
* Оптимальное назначение регистров (Register Allocation)
* Переупорядочивание блоков
* Предвыборка данных и кода
* Применение векторных инструкций
* Развертывание циклов (Loop unrolling)
* Оптимизация прологов и эпилогов функций

*Наиболее важные оптимизации*

* Скаляризация (scalarizing)
  + Разложение массивов и объектов на локальные переменные
  + Уменьшает затраты на выделение памяти и сборку мусора
  + Улучшает локальность данных
* Девиртуализация (devirtualization)
  + Замена виртуального вызова на прямой вызов
  + Уменьшает затраты на косвенные вызовы
  + Необходима для подстановки
* Специализация и клонирование методов (specialization and cloning)
  + Разделение методов по типам параметров
  + Требуется для межпроцедурной оптимизации
* Подстановка (inlining)
  + Подстановка тела метода вместо вызова
  + Подстановка констант
  + Уменьшение затрат на вызовы
  + Улучшает локальность кода и данных