JANCY: Возвращение Указателей

Contents

[Введение 1](#_Toc374884267)

[Принципы дизайна 1](#_Toc374884268)

[Ключевые возможности 1](#_Toc374884269)

[Другие значимые особенности 2](#_Toc374884270)

[Источники вдохновения 2](#_Toc374884271)

[С 2](#_Toc374884272)

[С++ 2](#_Toc374884273)

[Java/C# 3](#_Toc374884274)

[D 4](#_Toc374884275)

[Мотивация 5](#_Toc374884276)

[Мотивация №1, велосипедная 5](#_Toc374884277)

[Мотивация №2, практическая 5](#_Toc374884278)

[Мотивация №3, философская 6](#_Toc374884279)

[Огласите весь список, пожалуйста 6](#_Toc374884280)

[Архитектура компилятора 8](#_Toc374884281)

[Заключение и статус 9](#_Toc374884282)

[Приложения 10](#_Toc374884283)

[Полная грамматика Jancy 10](#_Toc374884284)

# Введение

Что?! Ещё один язык программирования? Я знаю, как это может звучать, но... Да. Знакомьтесь, это Jancy. С самого начала приведу список особенностей языка, который поможет вам определиться, а стоит ли вообще продолжать чтение статьи.

### Принципы дизайна

* Объектно-ориентированный язык для системного программирования с C-подобным синтаксисом
* Использование как скриптового языка из C/C++ приложения
* ABI (application-binary-interface) совместимость с C/C++
* Автоматическое управление памятью через точную сборку мусора (accurate garbage collection)
* LLVM в качестве backend

### Ключевые возможности

* Безопасные указатели и адресная арифметика
* Свойства (properties) – одна из самых полных реализаций
* Парадигма RAII (resource-acquisition-is-initialization)
* Реакционное программирование (reactive programming) – одна из первых реализаций в императивном языке программирования общего назначения

### Другие значимые особенности

* Замыкания (closures) для функций и свойств
* Слабые указатели (те, которые не влияют на время жизни объекта)
* Мультикасты и слабые мультикасты (те, от которых не обязательно отписываться)
* Явное выделение со стека, ручное управление памятью, локальная память потоков (thread local storage)
* Const-корректность
* Множественное наследование
* Perl-подобное форматирование строк
* …и многое другое – нет, правда! ;)

Теперь, если мы всё ещё вместе, позволю себе немного замедлиться и обернуться назад.

# Источники вдохновения

### С

Давным-давно, в далёкой-далёкой галактике... а если конкретнее, в конце шестидесятых в AT&T Bell Labs Деннисом Ритчи был создан самый главный язык программирования всех времён и народов: язык C. Значение его трудно переоценить. Даже сейчас, спустя больше 40 лет после появления, старый добрый C остаётся одним из наиболее используемых языков – независимо от выбора конкретного рейтинга популярности/используемости языков программирования. Причём первые места на вершине любого подобного рейтинга он будет делить со своим непосредственным потомком C++ и другим родственными языками, синтаксис которых был порождён из синтаксиса С: Java, Javascript, C#, Objective C. Если же рассматривать только использование в системном программировании, то лидирующий отрыв C и С++ от остальных языков будет ну просто неприличным.

Я люблю C. Синтаксис C прост, элегантен, выразителен и универсален. И, что очень важно в системном программировании, прослойка кода, генерируемая компилятором, тонка и предсказуема: опытный программист, взглянув на кусочек кода на C может сразу приблизительно прикинуть как этот же кусочек будет выглядеть на ассемблере после компиляции.

### С++

Появившийся спустя десятилетие после C его прямой наследник C++, к сожалению уже не может похвастаться этими качествами. У языка C++ есть горячие сторонники и не менее горячие непримиримые противники. Священные войны, посвященные C++, нередки на любом программистском форуме в любое время года. Споры обычно строятся вокруг следующих аргументов.

* C++ действительно обладает крайне широкими выразительными возможностями
* Программы, написанные на С++, отличает высокая производительность

Второе тесно связано с первым. Во многом именно вследствие выразительности синтаксиса разработчик может создавать высокоэффективный код.

* C++ сложен!

Сложен настолько, что абсолютно справедливо заслужил себе место в программистских шутках. Вместе с замечательными выразительными возможностями язык предоставляет многочисленные и разнообразные возможности «выстрелить себе в ногу». Написать плохой код на C++ гораздо легче чем на написать плохой код на C. Кроме того, неявно генерируемый компилятором код больше, требует больше памяти, и его уже не так легко предсказать даже опытным разработчикам.

Вполне вероятно, что именно этими причинами и обусловлен тот факт, что даже после появления хороших компиляторов C++ язык C остался главным языком системного программирования. Windows, Linux, Doom и Quake написаны на C в то время как C++ существовал уже многие годы.

* В С++ отсутствуют многие возможности, являющиеся встроенными в большинство современных языков, такие как автоматическое управление памятью или reflection.

Подведём итог: эффективно программировать на C++ безусловно можно, однако требуется высокая квалификация разработчика. Кроме того, даже квалифицированный специалист временами стреляет в ноги себе и коллегам.

### Java/C#

На защиту ног разработчиков от случайных выстрелов пришли managed языки. Спустя ещё одно десятилетие после создания C++ появилась Java. Созданный Microsoft язык C# изначально выглядел как клон Java, бездумно копирующий и хорошее и плохое; впоследствии, однако, C# был весьма удачно и остроумно расширен. В дальнейшем я буду рассматривать только Java, но подавляющая часть всего, о чем будет сказано ниже, в той или иной мере относится и к C#

Компилятор Java генерирует не машинный код для конкретного процессора, а Java байт-код, который затем интерпретируется или JIT-компилируется виртуальной машиной Java. Это позволяет использовать один и тот же бинарный модуль на различных архитектурах без перекомпиляции.

Помимо этого, виртуальная машина Java обладает встроенным механизмом автоматического управления памятью через точную сборку мусора. Разработчикам на Java не нужно беспокоиться о явном удалении выделенных под объекты блоков. Это, конечно, не означает что утечки памяти в Java невозможны. Кроме того, иногда желательно управлять памятью вручную, а не полагаться на удобный, но недетерминированный механизм сборки мусора. Но в целом наличие встроенного сборщика мусора это неоспоримый плюс, который признают большинство адекватных сторонников C/C++:

* Значительная порция кода C/C++ программы посвящена освобождению выделенной памяти (в частности, подавляющая часть деструкторов C++ классов становится просто не нужна, так как занимается исключительно освобождением памяти)
* Альтернативные механизмы автоматического управления памятью, доступные C++ разработчикам (а именно различные вариации подсчёта ссылок или консервативные сборщики мусора) во многих отношениях уступают встроенному точному сборщику мусора (accurate gc)
* Передача объектов между С++ модулями в значительной мере затруднена из-за отсутствия стандартизованного механизма освобождения памяти
* Исчезает целый класс багов, связанных с преждевременным удалением или напротив, неудалением блоков памяти

Вследствие вышесказанного, сборщик мусора упрощает, а значит, ускоряет и удешевляет процесс разработки.

Синтаксис Java представяет собой сильно модифицированный в сторону упрощения C++: синтаксис Java может быть выражан контекстно-свободной LR (1) грамматикой – в отличе от контекстно-зависимой грамматики C++, которая считается одной из (если не самой) сложной грамматикой из всех существующих языков программирования.

Помимо упрощения грамматики, под волну сокращений также попали:

* Пространства имён
* Псевдонимы типов (typedef)
* Глобальные функции и переменные
* Указатели и адресная арифметика
* Структуры (struct) и объединения (union)
* Явное создание экземпляров классов на стеке, RAII и вообще ручное управление памятью
* Перегрузка операторов
* Множественное наследование
* const-корректность
* Умолчальные значения аргументов

Ни в коей мере не желаю развязать очередную священную войну, рискуя всё-таки сказать, что большинство сокращений были незаслуженными. Да, возможно, обоснованными: ради упрощения синтаксиса/компилятора/виртуальной машины и ради повышения общей безопасности языка за счёт отъёма у программистов «опасных» инструментов. Но в то же время легко привести примеры, в которых использование той или иной возможности из этого списка позволяет сделать код иногда логичнее, иногда элегантнее, и часто эффективнее.

В целом же автоматическая сборка мусора, знакомый С-подобный синтаксис, безопасность и возможность работы собранного бинарника под любой платформой обусловили несомненный успех Java. В упомянутых в самом начале статьи рейтингах популярности языков Java стабильно занимает лидирующие позиции и более того, улучшает их от года к году.

### D

Несмотря на то, что в тех же рейтингах язык программирования D зачастую даже не попадает в десятку-двадцатку лидеров, было бы несправедливым не сказать хотя бы пару слов про этот интересный язык. Созданный Уолтером Брайтом как переосмысление C++, язык D появился приблизительно в одно время с C#. Основным мотивом в дизайне D стало объединение двух миров: производительность мира С/C++ и удобство managed мира.

D поддерживает полную совместимость с ABI (application binary interface) С/С++ программ, предоставляет указатели на данные и функции, позволяет делать вставки на ассемблере, и в то же время предоставляет такие вкусности из managed мира, как автоматическое управление памятью, вложенные функции и замыкания, плюс предлагает сильные возможности метапрограммирования и многое другое.

Одно время я серьёзно рассматривал использование D, как скриптового языка в наших приложениях, но молодость языка, а главным образом отсутствие на тот момент доступного для коммерческого использования backend-а, заставили отказаться от этой идеи.

Кроме того, ряд моментов в дизайне довольно сомнительны (напр, scope guards, или же семантическая трактовка функций как свойств), а небезопасные указатели или ассемблерные вставки в значительной мере закрывают возможность использования D как безопасного скриптового языка.

# Мотивация

### Мотивация №1, велосипедная

Одним из замечательных свойств российского образования было и, надеюсь, остаётся, привитие критического взгляда на вещи. Начиная со школы и, конечно, в институте, нас учили не брать слепо на веру то, что пишется в книжках или говорится лектором. Помню, на самой первой лекции по физике нам с серьёзным видом доказали что 2 = 5, и сразу после этого – что тупой угол равен острому. Затем лектор озвучил свою мысль: старайтесь не выучить то что сделано или сказано умными людьми (имея в виду, очевидно, себя), а осмыслить и понять почему сделано именно так, а не иначе. Возможно, вы найдёте ошибки в их рассуждениях. Возможно, вы найдёте способ сделать то же самое лучше, проще, элегантнее. А возможно и наоборт, вы поймёте что лучше приложить свои усилия в другом месте, а не на почве улучшения чего то, что работает и так.

С одной стороны, я безумно благодарен за привитый тип критического мышления. С другой стороны, из меня в определённой мере получился неисправимый изобретатель велосипедов – а это обычно не то звание, которым следует гордиться. В свою защиту скажу, что другое правило, привитое в институте – правило изучать типоразмеры уже существующих велосипедов перед изготовлением своего собственного, – несомненно сократило количество сделанных мной велосипедов на порядок.

Я занимался программированием в течение 20 последних лет, из них 15 – профессионально. В каждом из языков, на которых мне приходилось писать, мой внутренний изобретатель велосипедов непременно искал и находил вещи, которые можно сделать лучше. Так получилось, что областью моей специализации стало системное программирование, в частности создание компиляторов. Поэтому неудивительно, что уже много лет у меня были свои игрушечные языки как полигоны для экспериментов. В свете моего признания в любви к языку C, также неудивительно что синтаксис в них чаще всего был C-подобным.

До недавнего времени, однако, названия у этих языков не было – несмотря на то что предыдущие версии (а скорее, инкарнации, т.к. при переходе к новой версии от старого кода, как правило, не осталавалось ни строчки) использовались в коммерческих проектах как DSL (domain-specific languages). Однако в какой то момент я решил что это не дело, и, после сравнительно недолгой лингвистической одиссеи нашёл нужное имя: Jancy.

Как несложно догадаться, Jancy это акроним: [in between] JAva aNd C. Можно сказать, что за основу была взята Java и я постарался привнести в неё те возможности, которые я люблю в C/C++. Или же наоборот, что за основу был взят C/C++ и обучен приёмам Управляемых Языков. Плюс, конечно же, я добавил в язык фишки, ещё пока нигде не реализованные (или, если скромнее, то возможно и реализованные, но пока не получившие распространения в mainstream-языках) – фишки о которых я всегда мечтал и которые успешно прошли полигоны моих игрушечных языков.

Итак, должен признать, что одним из главных мотиваторов создания Jancy было насыщение моего внутреннего изобретателя велосипедов.

### Мотивация №2, практическая

Однако было и практическое обоснование для начала полномасштабных работ. Несколько лет назад наша компания выпустила продукт под названием I/O Ninja, который, в частности, выполнял анализ сетевых пакетов и должен был записывать в лог результаты анализа. Хочется избежать несправедливых обвинений в рекламе, поэтому я намеренно не расписываю подробности функционала, и зачем вообще такое потребовалось. Будем исходить из предположения, что нам надо было анализировать и генерировать бинарные пакеты. Так как анализ хотелось сделать максимально гибким, надо было вынести логику анализа либо в плагины, либо в скрипт. А так как в идеале мы хотели дать пользователям возможность дописывать свои собственные анализаторы/модифицировать существующие, скрипт выглядел как более правильный подход (плагины означали бы необходимосить публикации SDK для плагинов, плюс наличия у клиентов средств разработки на С/C++ и людских ресурсов с необходимым уровнем квалификации, чтоб эту разработку вести)

Проблема состояла в том, что разбор бинарных данных в языке без указателей и возможности описания структур и объединений (struct/union) сводится к вытаскиванию байтов по фиксированным индексам из буфера и складыванию их в слова, что порождает нечитабельный и неподдерживаемый код. Для работы с бинарными данными (анализа или генерации) нужны структуры, указатели и адресная арифметика. Точка. Общепризнанных скриптовых языков для встраивания в C/C++ приложения с поддержкой указателей или какого то другого хитрого механизма для работы с бинарными данными не нашлось ни несколько лет назад, ни сейчас, в момент написания статьи. Значит, если они и существуют, то только в экспериментальных вариантах, и любой самодельный велосипед имеет не меньшее право на жизнь.

Первая версия I/O Ninja вышла-таки на архитектуре плагинов, вторая – на скриптах, написанных на прототипе Jancy (опасные указатели, управление памятью на основе подсчёта ссылок, рудиментарная поддержка классов и самописная виртуальная машина). Третья версия находится в состоянии активной разработки и основана на полноценной Jancy.

Этот продукт – практический мотиватор создания Jancy и прекрасный полигон для тестирования языка в реальных условиях.

### Мотивация №3, философская

Перед тем как перейти к техническим подробностям и полному списку возможностей языка Jancy, позволю себе ещё немного пофилософствовать. Я ни в коей мере не хочу сказать, что создания языка было «вынужденным». Что без него невозможно было реализовать всё то же самое на других языках или других технологиях. Можно было. Всегда есть альтернативный путь.

Но так работает прогресс: большинство нововведений просто делают некоторый доступный уже человечеству процесс немного более удобным, эффективным, лёгким и т.д. Можно было бы вообще всё написать на старом добром С. И даже прямо на ассемблере! Как можно и перемещаться из пункта А в пункт Б на лошадях, или освещать помещение керосиновой лампой. Но любой процесс можно чуть-чуть улучшить. А потом ещё чуть-чуть улучшить. Из таких маленьких улучшений, каждое из которых отнюдь не претендует на революционность и не открывает по-настоящему новых возможностей (то же самое ведь можно было сделать и раньше, только по-другому!) – складываются большие: такие, которые позволяют сделать что-то, ранее принципиально невозможное. Но всё всегда начинается с маленького инкрементального улучшения.

Этими маленькими инкрементальными улучшениями мы и пытаемся заниматься.

# Огласите весь список, пожалуйста

* Полная ABI-совместимость с хостовым приложением на C/C++
  + Правильно описав типы данных в хостовом приложении и jancy скрипте, можно напрямую передавать данные через аргументы и возвращаемые значения, без необходимости явного заталкивания на стек виртуальной машины, упаковки в variant-подобные контейнеры и т.д. Поддерживаются
    - Все примитивные типы C/C++ (а также целочисленные типы с обратным порядком следования байтов)
    - Структуры (с произвольным фактором упаковки)
    - Объединения
    - Массивы
    - С/С++ указатели на данные и функции
* Классы (представляют собой особый тип данных со специальным заголовком)
  + Поддерживаются стандартные для современных языков возможности, такие как
    - конструкторы и деструкторы (статические и нестатические)
    - инициализиция полей по месту объявления
    - перегрузка операторов
    - виртуальные функции
  + Простое множественное наследование (без виртуальной модели C++)
  + Реализация методов может располагаться как по месту объявления, так и быть вынесена за пределы класса (как в C++)
  + Поддерживаются т.н. преконструкторы (preconstruct) – функции, которые будут автоматически выполнены перед каждым из перегруженных конструкторов. Аналог инициализирующих блоков в Java
* RAII, а также полный контроль над размещением данных
  + Возможность явно выделять память под объекты со стека и использовать парадигму RAII (как в C++)
  + Возможность выделять память под множество объектов сразу (объявляя поля-эклемпляры классов как в C++), без необходимости явного вызова ‘new’ под каждый объект.
  + Встроенная поддержка потоковых переменных (thread)
  + Ручное управление паматью с помощью heapu new/delete
  + Спецификаторы размещения: static, heap, heapu, stack, thread – могут быть указаны как при объявлении полей и переменных, так и при использовании оператора new
* Указатели на данные и классы
  + Const-корректность
  + Безопасная адресная арифметика
  + Встроенная поддержка слабые указателей на классы
* Свойства (без ложной скромности – самая полнофункциональная на текущий момент реализация)
  + Естественная форма объявления простых свойств
  + Полная форма для объявления свойств произвольной сложности (с перегруженными сеттерами, полями данных, вспомогательными методами и тд)
  + Индексируемые свойства (свойства, имеющие семантику массивов)
  + Autoget-свойства избавляют от рутинной необходимости прописывать геттеры-дегенераты
  + Bindable-свойства (свойства с событием onChange, т.е. оповещающие о своём изменении)
  + Bindable-данные для автоматической генерации свойств, единственным назначением которых является отслеживание изменений.
* Указатели на функции и свойства
  + Указатели на функции и свойства умеют захватывать значения аргументов
  + Автоматическое создание функций-переходников при несоответствии сигнатур аргументов
  + Слабые указатели на функции и свойства
  + Оператор планировки (schedule): создание указателя на функцию, которая будет вызвана в нужном контексте
* Мультикасты и события
  + Мультикасты позволяют накопить массив указателей на функции и затем вызвать их все сразу
  + События это особый вид указателя на мультикаст, который позволяет только подписаться/отписаться, но не вызвать
  + Два вида преобразовния мультикаста к указателю на функцию: live и snapshot
  + Встроенная поддержка слабых мультикастов (то есть тех, для которых не нужен явный вызов Unsubscribe)
* Поддержка реакционного программирования
  + Дилемма сосуществования реакционного и императивного кода разрешена введением специальных зон реакционного кода, т.н. «реакторов»
  + Выражения внутри реактора автоматически пересчитываются при изменении bindable-свойств, входящих в правую часть
  + Специальные конструкции для назначения блоков кода как обработчиков изменения заданных bindable-свойств (без необходимости явной подписки/отписки).
* Специальная модель обработки исключений
  + Исключения в Jancy это просто синтаксический сахар над стандартной моделью кодов ошибок
  + Как следствие, модель исключений Jancy прозрачна и совместима с любым внешним языком
  + catch и finally могут быть объявлены в любом блоке
  + При вызове одной и той же функции можно применять как проверку кода ошибки, так и исключения – в зависимости от того, какой подход более удобен в данном конкретном случае.
* Дуальная модель доступа
  + В Jancy только два спецификатора доступа: public и protected. Для каждого пространства имён весь остальной мир распадается на «своих» и «чужих». Свои имеют доступ ко всему, чужие – только к public
  + Дуальный модификатор constd позволяет объявить тип данных, который выглядит как const для «чужих», и в то же время доступен для изменения «своими». Предназначен для избавления от необходимости прописывать свойства-дегенераты, единственное назначение которых это предоставление доступа только-на-чтение.
  + Дуальный модификатор eventd для объявления событий или указателей на события. «Чужие» могут только подписатья/отписаться (семантика «event»), свои же должны уметь вызвать всех подписчиков (семантика «multicast»)
* Специальные литералы
  + Hex-литералы для удобного описания блоков константных двоичных данных
  + Форматирующие литералы позволяют динамически генерировать строки с подстановкой значений выражений, используемых внутри литерала (perl-style formatting)
* Другое
  + once/thread once
  + Полноценные curly-инициализаторы
  + break-n/continue-n для управления внешними циклами
  + basetype-n для удобной адресации базовых типов
  + enumf для описания флаговых перечислений
  + extend – добавление методов в существующие типы
  + alias – псевдонимы выражений
  + Scopes-in-switch
  + Конструкторы/деструкторы у юнитов компиляции

# Архитектура компилятора

Для генерации лексического анализатора используется Ragel – универсальный компилятор конечных автоматов, который замечательно подходит для создания лексеров в силу удобства входного языка и эффективности выходного кода – затыкающей, кстати говоря, за пояс все аналоги.

В качестве backend используется LLVM, что позволяет задействовать отлаженный оптимизатор и кодогенериратор сразу для широкого спектра платформ, а также существенно упрощает обеспечение совместимости с ABI хостового C/C++ приложения. Использование LLVM, а не рукописной виртуальной машины, или же рукописного генератора целевого кода – это решение которое не вызывало вопросов с самого начала.

А вот для синтаксического анализа я таки не удержался и написал свой ~~велосипед~~ генератор табличных нисходящих LL (k) анализаторов. Чем меня не устраивали ANTLR, Coco, Yacc/Bison, Lemon и другие безусловно уважаемые и проверенные в бою парсер-генераторы, это тема для отдельного и интересного разговора. Пока же я просто хотел сказать что в качестве синтаксического анализатора в компиляторе Jancy используется не ручной рекурсивный спуск, а сгенерированный парсер, что помимо прочего означает, что имеется постоянно-релевантная BNF грамматика которую можно распечатать, обсудить и с лёгкостью модифицировать. Грамматика Jancy относится к классу контескстно-зависимых LL (2).

Выходом парсера является сразу LLVM IR без промежуточной генерации AST: в нисходящих парсерах удобно проводить семантический анализ и генерировать код прямо по ходу разбора.

Модель взаимодействия лексера и парсера подсмотрена в Lemon: парсер Jancy НЕ вызывает лексер; вместо этого имеется внешний цикл выборки токенов, каждый из которых скармливается табличному парсеру. Данная модель позволяет разбирать неполные юниты компиляции, останавливать и возобновлять процесс разбора и т.д., плюс, как и в любом табличном парсере, память под аттрибуты правил выделяется не на стеке (что чревато stack overflow или искусственными ограничениями на уровень вложенности), а в куче.

Синтаксический/семантический анализ мультипроходный (как правило, два прохода, в особых случаях – три), что, однако, не означает повторного запуска лексера. Второй проход сделан для возможности использовать глобальные типы и данные до парсинга их объявления (которое может находиться вообще в другом юните компиляции). Третий проход необходим, в частности, для предварительного рассчёта реакторных классов.

# Заключение и статус

Jancy это кросс-платформенный проект, нацеленный в первую очередь на Windows, Linux и Mac (порт на Mac пока не реализован); компилятор способен выдавать целевой код для любой архитектуры процессоров, поддерживаемой LLVM. Jancy не является проектом с открытым исходным кодом, однако может стать таковым в дальнейшем. Jancy-плагин для NetBeans с поддержкой code-assist и отладки через GDB находится в состоянии активной разработки и, скорее всего, будет выпущен одновременно с новой версией I/O Ninja.

В настоящий момент на сайте нашей компании доступна тестовая страничка компилятора Jancy – без необходимости что-либо скачивать и устанавливать. Определённые возможности, которые можно ожидать от современного языка (такие как reflection, generics/templates, лямбда-функции и т.д.) пока не реализованы, но обязательно появятся в будущих релизах. В то же время возможности, которые, как мы считаем, уже готовы и отлажены, после ударного тестирования через веб, обязательно обнаружат недоделки, недоработки и просто элементарные баги. В конце концов, это первый публичный релиз, и баг-репортам мы будем не менее рады чем положительным отзывам. Мы уверены что во время разработки новой, основанной на Jancy, версии I/O Ninja и с помощью онлайн-тестирования, нам удастся как следует обкатать язык и компилятор и сделать их пригодными для применения в широком классе приложений.

Мы надеемся, что нам удалось пробудить интерес к проекту и мы будем счастливы получить от уважаемого сообщества разработчиклов отзывы любой степени доброжелательности, вопросы и предложения.

# Приложения

### Полная грамматика Jancy

axl\_jnc\_Lexer.rl

axl\_jnc\_Decl.llk

axl\_jnc\_DeclarationSpecifier.llk

axl\_jnc\_Declarator.llk

axl\_jnc\_Expr.llk

axl\_jnc\_Expr\_s.llk

axl\_jnc\_NamedTypeSpecifier.llk

axl\_jnc\_Parser.llk

axl\_jnc\_Stmt.llk

axl\_jnc\_Stmt\_0.llk