1. Титульный
2. Общие требования
3. Характеристики фронтендов и генераторов
4. Требования к инструменту, пользователь
5. Архитектура=dataflow
6. Реализованные компоненты(3+2+3)
7. Реализация(CLI, автозагрузка dll, тестирование)
8. Возможные цепочки, юзкейсы(рек.сп->GLR)
9. Технологии

Здравствуйте! Меня зовут Константин Улитин, я студент кафедры системного программирования мат-меха спбгу. Тема выступления – модульный генератор синтаксических анализаторов. Разрабатываемое нами приложение имеет название YaccConstructor, которое упоминалось(будет упоминаться) и в других работах. Я расскажу, чем оно отличается от других генераторов СА. (И, несмотря на название, с прародителем всех Г.С.А. yacc-ом оно связано только косвенно).

Требования, выдвигаемые к Г.С.А., отличаются широтой и разнообразностью, и часто выполнение одного затрудняет выполнение другого. /\*Среди основных выделяются класс принимаемых языков и грамматик, разрешение неоднозначностей, скорость работы сгенерированного транслятора, удобство описания трансляции, в частности, работа с L-атрибутами, простота отладки, восстановление после ошибок, целевой язык и так далее.\*/ На слайде вы видите наиболее значимые. Возможно, широта требований объясняет такое большое число существующих на данный момент Г.С.А. Однако наш инструмент пытается не заменить, а дополнить другие позволяя использовать преимущества того или иного инструмента.

Рассмотрим поподробнее, какими характеристиками инструментов достигается выполнение этих требований. Многие из них завязаны на выбранный класс алгоритмов разбора. Например, LL-анализатор с алгоритмом рекурсивного спуска удобен для отладки грамматики, но сгенерированный (или вручную написанный) транслятор обычно медленнее табличных. И такой анализатор не сможет работать с леворекурсивными грамматиками. LR и LALR, в отличие от него, могут работать с леворекурсивными, но опять же существуют общеизвестные проблемы, например shift-reduce конфликт if-then-else. И леворекурсивность, и конфликтность грамматики решаются путем ввода новых искусственных правил, но при этом ухудшается читаемость, и следовательно сопровождаемость грамматики. GLR-анализатор позволяет работать с неоднозначными грамматиками, поэтому лишен этих недостатков. Немалое значение на удобство разработки и возможность переиспользования реализованных частей(в частности, при разработке диалектов) оказывают и возможности языка описания трансляции, такие как конструкции РБНФ, макроправила , модульность грамматики.

// DSL, конфигурационные файлы, собственные протоколы; компиляторы, статический анализ, IDE

Соответственно, была поставлена задача, с одной стороны, дать возможность пользователю самому выбирать, в каком формате у него будет описана грамматика, и какой алгоритм будет использовать порожденный транслятор, с другой стороны, постараться вместить максимальное количество возможностей в один из языков описания. Как будет видно дальше, предложенная архитектура это позволяет, и даже отчасти требует (чтобы поддерживать использование максимального числа сторонних инструментов).

Инструмент построен вокруг унифицированного представления грамматики. Соответственно, парсер входной грамматики на входе имеет текст или что-то другое с ее описанием, а на выходе – объект заданного нами класса. Затем с этим объектом могут производиться необходимые преобразования, после чего он поступает на вход генератору. Компоненты реализуют соответствующие интерфейсы, указанные на диаграмме.

На данный момент реализованы следующие компоненты – фронтенды Yard(собственный язык описания трансляций с поддержкой ...правил, параметризованных другими правилами(что-то похожее на шаблоны в C++), конструкций EBNF; ANTLR(преобразует описание трансляции в формате antlr-a в наше внутреннее представление), Irony(Г.С.А на C#, в котором грамматика задается классом, а не текстовым файлом). Генераторы – RACC(про него сегодня рассказывал Семен Григорьев), FParsecGenerator(печатает грамматику в формате этого инструмента), YardPrinter(печатает в формате нашего инструмента). Преобразования – ExpandMeta(подставляет в шаблоны правил другие правила), ExpandEBNF(раскрывает конструкции РБНФ - ?,+,\*).

Работа с приложением идет через интерфейс командной строки и позволяет пользователю выбирать фронтенд и генератор, просматривать список доступных. Компоненты собираются отдельно от главного модуля, соответственно представляют из себя DLL-ки. Главный модуль умеет их подключать без перекомпиляции, надо лишь подложить их в ту же директорию. К большинству компонент написаны unit-тесты.

/\*Пусть у нас есть описание трансляции для языка COBOL, написанное на Yard, мы хотим в него добавить поддержку встроенных SQL-выражений. Для инструмента ANTLR уже есть описанная грамматика SQL. Поэтому мы подкладываем этот файл рядом с Cobol.yrd, в Cobol.yrd добавляем строчку include “sql.g” и с минимальными доработками получаем то, чего хотели. \*/

Рассмотрим несколько возможных случаев использования YaccConstructor-a. Пусть разработчик хочет модифицировать грамматику SQL, уже написанную на ANTLR. Он не знает синтаксиса ANTLR, и у него нет времени с ним разбираться. Тогда он берет эту грамматику и транслирует ее в yrd, выбрав в качестве фронтенда AntlrFrontend, а генератора – YardPrinter. Или например мы разрабатывали грамматику для LALR-анализатора, но в некоторый момент грамматика перестала быть LALR. Тогда мы переключаем генератор на другой, охватывающий более широкий класс, например LR(1). Третий случай использования – разрабатываем и отлаживаем грамматику на инструменте с алгоритмом разбора рекурсивным спуском, а выпускаем на GLR, так как он как правило быстрее работает.

Приложение разрабатывается под платформу .NET, основной используемый язык – F#. Используемая среда разработки – Visual Studio 2010. Исходный код и некоторую документацию можно найти в репозитории проекта по указанному адресу. Или ввести в поиск YaccConstructor☺