Оглавление

Введение

Цель.

Этот проект появился как полувековая мечта разобраться с трансляторами, компиляторами и интерпретаторами, лексикой, синтаксисом и семантикой, левым и правым, нисходящим и восходящим, яками и бизонами.

Это - лабораторки по трансляторам. Результат прочтении «Книги дракона» в трех изданиях «зеленом», «красном» и «фиолетовом».

Ключевые слова.

LL1, LL(k), LR0, LR1, LR(k), SLR(k), LARL, LALR(k), YAAC, Core, `Closure`.

Грамматики

В проекте используется несколько грамматик, различающихся по некоторым признакам.

Есть грамматики описания учебного C – подобного языка (LL, NL, LR, NR) и грамматики учебные примеры (ETF_LL, ETF_LR, …).

Леворекурсивные грамматики (LR, NR), праворекурсивные грамматики (LL, NL).

Для упрощения описания грамматики используется два режима работы сканера: простые лексемы (NL, NR) и групповые лексемы (LL, LR). В первом случае сканер разбирает только одну лексему и отдает ее на выход. Во втором случае могут анализироваться несколько лексем и на выход отдается одна лексема, но сформированная на основе соседних, или групповая лексема, собранная из нескольких.

Для грамматик (LL, NL, LR, NR) условной компиляцией задается объем грамматики:

- Выражения
- Операторы и выражения
- Определения без функций
- Полная версия грамматики

Кроме синтаксических правил в грамматиках определены различные Действия.

- Tree – построение синтаксического дерева.
- Code – непосредственная генерация кода стековой машины.
- Node – построение в стеке разбора дерева типизированных узлов.
- Turn – построение дерева узлов универсального типа.
- Made – непосредственное формирование в стеке разбора дерева (Tree).

Ниже приводится перечень грамматик, заданных макрокомандами в тексте программы:

- 0x00, LL – праворекурсивная грамматика с групповыми лексемами, с действиями по формированию синтаксического дерева, команд стековой машины и дерева узлов (Tree, Code, Node).

- 0x01, NL – праворекурсивная грамматика с простыми лексемами, с действиями по формированию синтаксического дерева (Tree). Данная грамматика строилась на основе грамматики LL с переходом от групповых лексем к простым. Процесс этот не завершен и грамматика не удовлетворяет многим требованиям. Она используется в качестве примера генерации программ разбора (PROG) и метода разбора, основанного на правилах (RULE).
- 0x02, LR – леворекурсивная грамматика с групповыми лексемами, с действиями по формированию синтаксического дерева и команд стековой машины (Tree, Code, Turn).
- 0x03, NR – леворекурсивная грамматика с простыми лексемами, с действиями по формированию синтаксического дерева, команд стековой машины и дерева узлов универсального типа (Tree, Code, Node). Служит основой для формирования таблиц YACC. Создана на основе грамматики LR с заменой лексем, так же, как и NL, не удовлетворяет некоторым правилам.
- 0x04, ETF_LL – грамматика (*,+,целое) без действий
- 0x05, ETF_LL – грамматика (*,+,целое) с действиями (Tree, Code)
- 0x06, ETF_LR – грамматика (*,+,целое) без действий
- 0x07, ETF_LR – грамматика (*,+,целое) с действиями (Tree, Code)
- 0x08, EXPR_LR – пример грамматики выражений от OrOr до Unary
- 0x09, EEE – грамматика (*,+,целое), использование ассоциативности и приоритетов

Таблица соответствия грамматики, метода разбора и действий.

	Make	PROG	Rule	LL(1), LL(k)	LR(0), SLR(k)		LR(1), LR(k)		LALR, LALR(k)				YACC
					Closure	Core	Closure	Core	Lookahead	Tab	Full	Core	
LL	Tree, Code	+	+	+	+	+	+	+	+	+	+	+	+
	Node, Made	-	+	+	-	-	-	-	-	-	-	-	-
NL	Tree	+	+	-	-	-	-	-	-	-	-	-	-
LR	Tree, Code	-	-	*	+	+	+	+	+	+	+	+	+
	Turn	-	-	*	+	+	+	+	+	+	+	+	-
NR	Tree, Code	-	-	-	+	+	+	+	+	+	+	+	+
	Node, Made	-	-	-	+	+	+	+	+	+	+	+	+

Примечание * - после операции удаления левой рекурсии.

Приоритеты и ассоциативность.

В грамматиках LL и LR используются лексемы Priority, которыми задаются обобщённые операции с приоритетом. Сканер, встретив операцию, генерирует соответствующую лексему данного типа. В учебном языке есть неоднозначность использования этих лексем. Проблема в использовании лексемы = в инициализации переменных. В этом случае должна генерироваться лексема <Oper,Add> а в других <Priority,Assign>. Лексемы выступают в качестве ключа в таблицах разбора, поэтому поиск в этих таблицах строится в два этапа. Ищем по исходной лексеме, затем выполняем некое приведение и ищем снова. Число попыток степень двойки от предпросмотров, поэтому при k > 1 операции присваивания не обобщаются.

Наличие лексем приоритетов позволяет использовать для грамматики LR оптимизацию ассоциативности.

Представление грамматик

Грамматики представляются несколькими способами. Основным представлением является структура в памяти (TGrammar), главным элементом которой является таблица формул. Таблица формул - это массив формул, которые в свою очередь массив продукций. А каждая продукция, это массив элементов. В качестве элементов выступают Терминальные символы, Нетерминалы и Действия.

Для удобства программирования в проекте введен единый элемент GSymbol.

Таблица формул либо формируется на основе исходного описания грамматики, либо восстанавливается из файлов специальной структуры. Исходное описание грамматики создается вручную или генерируется на основе таблицы внутреннего представления. Возможны следующие представления грамматики:

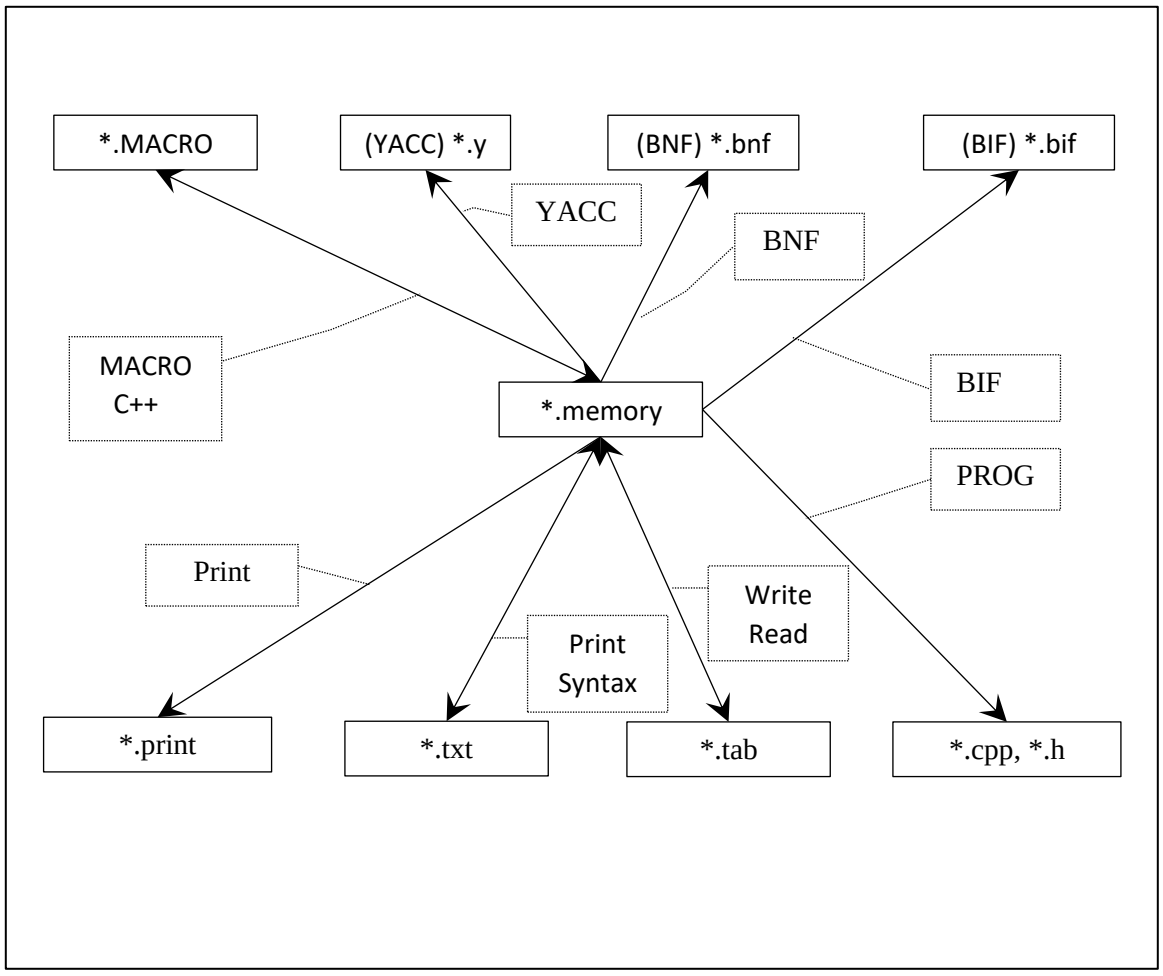
- *.memory - Таблица формул в памяти
- *.MACRO – Запись грамматики макрокомандами FORMULA и PRODUCT
- *.tab - Файл особой структуры, используемый для сохранения и восстановления грамматики.
- *.print – Печать грамматики в разных режимах.
- *.cpp, *.h – Программа на C++ сгенерированная по описанию грамматики, которая при выполнении генерирует таблицы грамматики в памяти. Модуль программ TranYACC и TableBuilder.
- *.y – Входной файл для YACC, генерируется методом YACC или создается вручную как первоисточник.
- *.bnf – Входной файл для BNF, генерируется методом BNF.
- *.bif – Входной файл для BIF, итерационного метода BNF, создается вручную как первоисточник, или генерируется методом BIF.
- *.tab.c – Выходной файл YACC. Модуль программы TranYACC.

Исторически первоисточником для грамматик (LL, NL, LR, NR) служит формат *.MACRO. Это требует перетрансляцию после внесения изменений, кроме того процесс трансляции довольно ресурсоемкий. Главным достоинством этого формата является наличие условной компиляции и вычисление выражений при компиляции.

Представление	Создание	Разбор	Примечание
*.memory	Метод make		
*.tab	Метод write	Метод read	
*.txt	Метод print	Table Builder	вручную
*.print	Метод print		
*.y	Метод YACC	YACC, YGrammar.y	вручную
*.bnf	Метод BNF	BGrammar.y	вручную
*.bif	Метод BIF	IGrammar.y	вручную

*.tab.c	YACC	C++	
*.MACRO	Метод MACRO	C++	вручную
*.cpp, *.h	Метод PROG	C++	

Есть возможность печатать разобранную грамматику в файл в форматах (*.y, *.bnf, *.bif) с возможностью ее разбора в новом формате. Можно закольцевать эти действия и тогда, при правильных разборе и печати, соответствующие тексты должны быть постоянными.



На основании таблицы грамматики могут быть построены другие таблицы, которые различаются в зависимости от метода разбора. Для любого табличного разбора доступны следующие:

- Таблица используемых лексем (*.lexeme)
- Таблица используемых действий (*.make)
- Таблица FIRST (*.first)
- Таблица EFF (*.eff)
- Таблица FOLLOW (*.follow)
- Таблица векторов для (к != 0)
- Таблица ссылок, позволяющие корректировать номера формул и таблиц, при изменении их размеров.

Для нисходящего разбора дополнительно строятся таблицы:

- Таблица SIGMA, используемая для проверки условия принадлежности грамматики к классу LL(k). Объединение строки для нетерминала равно строке FOLLOW.
- Таблицы LLk
- Таблица нисходящего разбора (*.cell)

Дополнительные таблицы для восходящего разбора:

- Таблица ситуаций (*.state)
- Таблица ядер (*.core), для варианта работы без замыканий. Для этого варианта предварительно строится таблица (*.c2a), указывающая на возможность вывода.
- Таблица распространения просмотров (*.look), для варианта LALR построения таблиц разбора
- Таблица переходов GOTO (*.goto)
- Таблица действий (*.action)

Выдача этих таблиц, в текстовом виде, побочное действие программы TableBuilder, основным является создание таблиц для программы разбора TranGrammar, куда входят соответствующая таблица разбора, таблица грамматики, таблица векторов для (к != 0) и другие. Эти таблицы выдаются либо в особом формате, читаемом программой TranGrammar, либо в виде вставки на C++ для формирования модулей для программ TranYACC и TableBuilder.

Разбор грамматик

Описание грамматики языка может быть записано по правилам другой грамматики, будем называть ее грамматикой разбора. Грамматика может не иметь машиночитаемого представления, ее разбор может осуществляться написанной вручную программой по методу прямого рекурсивного спуска.

Грамматики разбора, записанные в виде текстового файла, сами нуждаются в дополнительном этапе разбора.

Есть три синтаксиса записи грамматик:

- Запись по правилам программы YACC,
  - *.yacc - грамматика, записанная по правилам входного файла программы YACC.
  - *.y - грамматика, записанная по модифицированным правилам входного файла программы YACC. Основная модификация связана со способом задания действий. В первом случае это вставки на C, во втором вызовы программ;
- Нормальная форма Бэкуса, *.bnf;
- Форма Бэкуса с итерациями, *.bif;

Сам разбор синтаксиса грамматик, в свою очередь, представляется в виде грамматики.

- FGrammar.yacc – грамматика YACC. Разборщик – выходной файл программы YACC, в котором действия реализуются как макрокоманды. Вставка на C.
- FGrammar.y – грамматика YACC, записанная по модифицированным правилам грамматики YACC.
- BGrammar.y – грамматика BNF, записанная по модифицированным правилам грамматики YACC.
- IGrammar.y – грамматика BIF, записанная по модифицированным правилам грамматики YACC.
- IGrammar.bif – грамматика BIF, записанная по правилам Бэкуса с итерациями.
- IGrammarLLk.bif – грамматика BIF, записанная по правилам Бэкуса с итерациями.

Разборщики всех грамматик кроме первой – совокупность таблиц разбора, действия задаются вызовом программ действий, с соответствующими параметрами. Для описателей *.y эти действия записываются в форме NODE, а для описателей *.bif в виде формы TURN. Эти действия строят синтаксическое дерево универсальных узлов, с последующей генерацией таблиц грамматики.

Грамматики *.yacc и *.y - это грамматики LR(1), а IGrammar.bif - это грамматика LR(2), IGrammarLLk.bif в свою очередь грамматика LL(2).

Наличие грамматики разбора и запись на ней разборщика, требует некой рекурсии. Для того чтобы построить таблицы разборщика надо разобрать его описание.

Грамматика YACC.

Построение разборщика грамматик проводилось в два этапа.

На первом этапе методом прямого рекурсивного спуска была написана программа по разбору модифицированной грамматики YACC. Непосредственно при разборе строится таблица грамматики.

Параллельно был создан файл формата входного файла программы YACC, где действия заданы в виде вставок на C++, и представляют собой макрокоманды по созданию таблицы грамматики. После

прогона этого файла через программу YACC, был получен разборщик в виде файла C, который затем был оформлен как модуль MyYSyntax программы TableBuilder.

На втором этапе был создан файл модифицированного формата входного файла программы YACC, в котором, действия - это вызовы подпрограмм по созданию дерева узлов. Этот файл был обработан с помощью программ первого этапа. Сгенерированный разборщик может быть представлен как многократно загружаемая таблица или как модуль программы TableBuilder.

В правилах описания грамматики терминальные символы перечисляются полностью и задаются заранее определенными идентификаторами или символом в одинарных кавычках, что в нотации C является числом, поэтому нет коллизий при описании грамматики разборщика и разборщик может разбирать и генерировать сам себя.

Файл FGrammar.y преобразован в файл FGrammar.bnf, на основе, которого также созданы таблицы разборщика.

Грамматика BNF.

Файл BGrammar.y для разбора правил BNF, был получен модификацией файла FGrammar.y. Разборщик грамматики BNF представлен как таблицы разбора LR(2). Файл BGrammar.y может быть преобразован в файл BGrammar. bnf, из которого можно получить точно такие же таблицы, таким образом разборщик BNF может разбирать свое собственное описание.

В правилах описания грамматики терминальные символы заранее не перечисляются, а задаются абстрактным терминалом <Symbol>. Для того чтобы не было коллизий в грамматике разборщика, сканер должен работать в дух режимах, абстрактного символа и описателя действий. Кроме того, заводятся специальные метасимволы.

Грамматика BIF.

Разбор файлов грамматики BIF выполняет разборщик, сгенерированный грамматикой IGrammar.y, файл IGrammar.y может быть преобразован в файл IGrammar.bnf. Этот разборщик преобразует файлы формата *.bif. Итерационную часть формул разборщик заменяет на служебные вставки по следующим правилам.

[A] -> f1, где f1 вызов служебной формулы **F1 ::= A |** .

[A]… -> f1, где f1 вызов служебной формулы **F1 :: = A | F1 |** .

{A} -> f1, где f1 вызов служебной формулы **F1 ::= A** .

Есть тестовые задания, которые преобразуют грамматики (LL, LR, NR) в формат BIF с последующим их разбором.

Преобразование грамматик разбора.

Первоначально описатели грамматик YACC, BNF и BIF были созданы вручную по правилам грамматик *.y, файлы FGrammar.y, BGrammar.y и IGrammar.y соответственно. Действия в них заданы по правилам создание дерева узлов (Node), промежуточная информация хранится в самом стеке разбора. В качестве параметров в действиях используются номера элементов в продукциях, поэтому важна неизменность продукций при преобразовании грамматики. Если действие является последним элементом продукции оно переносится в заголовок продукции. Если действие внутри продукции,

создается служебная пустая продукция с действием в заголовке. Преобразование грамматики YACC в грамматику BNF производится автоматически, так как оно не изменяет структуру формул и продукций. Изменяются только форма представления элементов продукции и действий.

Режимы разбора.

Существуют режимы разбора описания грамматик, записанных по указанным правилам.

- 0x0E – разбор грамматики формата (*.y) методом прямого рекурсивного спуска; Таблица грамматики создается непосредственно при разборе.
- 0x0D – разбор грамматики формата (*.y) с помощью таблиц разбора, созданных по правилам FGrammar.y;
- 0x0C – разбор грамматики формата (*.y) программой, созданной системой YACC по описателю FGrammar.yacc; ; Таблица грамматики создается непосредственно при разборе. Действия заданы как макрокоманды по правилам системы YACC.
- 0x0B – разбор грамматики формата (*.bnf) с помощью таблиц разбора, созданных по правилам BGrammar.y;
- 0x0A – разбор грамматики формата (*.bif)) с помощью таблиц разбора, созданных по правилам IGrammar.y и по правилам IGrammarLLk. bif.
- 0x09 – разбор грамматики формата (*.*) с помощью таблиц разбора, читаемым из файла грамматики.

Разборщики режимов 0x0C и 0x0E представлены в виде текстов на C++, первый создан как программная вставка, результат работы системы YACC, а второй написан вручную. Эти разборщики использовались на первом этапе для построения других разборщиков табличного типа.

Разборщики режимов 0x0D, 0x0B и 0x0A – таблицы разбора для методов LR(0) и LR(1) в первом случае, и LR(2) для второго и третьего, так как их описатели представляют собой леворекурсивные грамматики. Описатели грамматик, записанные в формате (*.bif) в дальнейшем преобразуются в праворекурсивную грамматику, поэтому их разборщики могут быть созданы в виде таблиц для нисходящего разбора для метода LL (2).

Таблицы могут быть представлены разными способами, но для простоты дальнейшей работы, они формируются в виде программных вставок, инициализирующих эти таблицы. Эти вставки части модуля TableBuilder, поэтому после получения таблиц разборщиков необходимо перетранслировать соответствующие программы. Все эти вставки выделены в отдельный каталог Generated для возможности первичной трансляции.

Разборщики, которые оформлены по правилам системы YACC, в которой должны быть заданы используемые терминалы, поэтому они кроме правил разбора собственно грамматики, содержат и описание терминалов или токенов, заданное по особым правилам. Наравне с этим, для упрощения описателей грамматик, на основании информации, получаемой из таблиц сканера, генерируются внутренние таблицы описания терминалов.

Грамматики C и C99.

Описание грамматик C и C99 взяты в виде входных файлов системы Bison и Flex. Особенностью этих грамматик является то, что в них присутствуют две лексемы, соответствующие идентификатору: IDENTIFIER и TYPE_NAME. Что предполагает на этапе лексического разбора формирования таблиц блоков и таблиц определения идентификаторов. В ходе своей работы сканер, по информации из этих

таблиц, определяет тип лексемы: IDENTIFIER, TYPE_NAME, DECLARATOR или ENUMERATOR. Кроме поддержки таблиц блоков и идентификаторов необходимо корректировать состояние стека грамматического разбора.

Сканер, сгенерированный системой Flex, подключен, через переходник, к системному сканеру и может быть альтернативой не только при разборе текстов с синтаксисом грамматик C и C99, но во всех остальных случаях.

Есть три варианта грамматики для C и C99:

- C_Grammar.yacc – грамматика C для системы YACC и подключение ее к TranYACC, без учета типа лексем IDENTIFIER
- C_Grammar.y – грамматика C для генератора таблиц разбора и подключение ее к TranGrammar, с учетом типа лексем IDENTIFIER.
- C99_Grammar.y – тоже для C99.

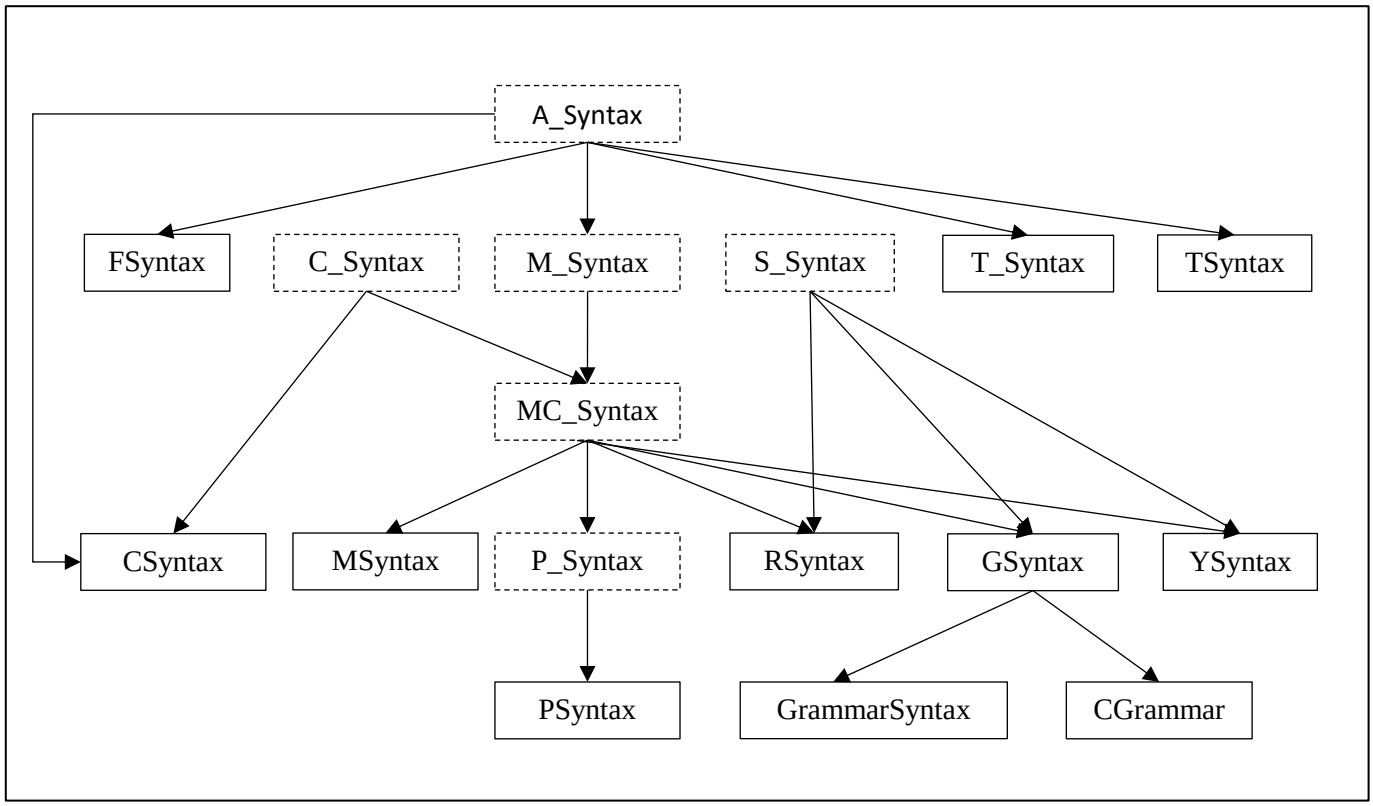
Варианты синтаксиса

Объектами синтаксического разбора являются:

- Тексты программ, написанных по правилам синтаксиса учебного C-подобного языка. Грамматика этого языка задана макрокомандами в виде грамматик (LL, NL, LR, NR). Есть возможность генерировать тексты грамматик в форматах BNF и BIF.
- Тексты программ, написанных по правилам синтаксиса учебных грамматик (ETF_LL, ETF_LR).
- Тексты описателей грамматик, записанные по правилам синтаксиса системы YACC.

Синтаксические анализаторы представлены в виде:

- Программ, написанных вручную (T_Syntax, TSyntax, CSyntax, MSyntax, FSyntax).
- Программ, интерпретаторов правил праворекурсивных грамматик (RSyntax).
- Программ, сгенерированных на основе праворекурсивных грамматик (PSyntax).
- Табличных автоматов нисходящего и восходящего разборов, сгенерированных по правилам грамматик (GSyntax).
- Программных вставок, сгенерированных системой YACC (YSyntax).



Виды программ синтаксиса:

- A_Syntax – абстрактный класс, отвечающий за связь со сканером.
- C_Syntax – абстрактный класс, реализующий аппарат генерации кодов*, использует* стек области видимости блоков и идентификаторов.
- S_Syntax – абстрактный класс, имеет стек разбора
- M_Syntax – абстрактный класс, реализация аппарата действий.
- MC_Syntax – абстрактный класс, действия по построению дерева разбора или генерации кода.
- P_Syntax – абстрактный класс, аппарат генерируемых программ прямого рекурсивного спуска.
- T_Syntax – проверка синтаксиса. Прямой рекурсивный спуск, программа пишется для каждой грамматики отдельно. Для каждой синтаксической формулы своя собственная функция разбора, возвращающая булевскую переменную.
- TSyntax – проверка синтаксиса и построение дерева, в дополнении к T_Syntax. Функции разбора возвращают узел синтаксического дерева.
- CSyntax - проверка синтаксиса и генерация кода, в дополнении к T_Syntax.
- MSyntax – проверка синтаксиса и построение дерева или генерация кода с помощью действий. Прямой рекурсивный спуск, программа пишется для каждой грамматики отдельно.
- RSyntax – интерпретация правил праворекурсивной грамматики. Нисходящий разбор с операциями возврата.
- GSyntax – распознаватели на основе таблиц конечного автомата с магазинной памятью. Нисходящий и восходящий разборы, с предпросмотрами и без. LL1, LLk, LR0, LR1, SLRk, LRk, LALR, LALRk. Есть восемь вариантов таких автоматов, определяемых методом (LL, LR), просматриваемыми символами (1, k) и видом стека сборки синтаксического дерева (стек разбора и внешний).
- YSyntax – аппарат связи с программными вставками, сгенерированными YACC.

- PSyntax – сгенерированные программы прямого рекурсивного спуска.
- CGrammar – реализация синтаксического разбора грамматик C и C99. Основная задача связь со сканером по формированию лексем TYPE_NAME, DECLARATOR, IDENTIFIER и ENUMERATOR. Проблема в том, что синтаксис рассчитан на использование двух лексем (TYPE_NAME и IDENTIFIER), тогда как сканер не может их различить самостоятельно. Поэтому при разборе приходится строить *Таблицу блоков* и *Таблицами идентификаторов* внутри сканера. Синтаксическое дерево в полном объеме не строится.
- GrammarSyntax - реализация разбора грамматик разбора методом таблиц восходящего или нисходящего разбора. Есть три варианта записи правил грамматики:
    - *.y – правила системы YACC
    - *.bnf – нормальная форма Бекуса
    - *.bif – итерационная форма Бекуса

  Для первого случая лексемы в описателе грамматики задаются заранее определенными идентификаторами, конструкциями вида <Группа> и <Группа, Тип> и целыми числами, в случае односимвольных. Поэтому нет коллизий при использовании метасимволов описания грамматики. Для двух последних вариантов лексемы в описателе грамматики задаются специальным метасимволом, что требует наличие у сканера состояний. Сканер настраивается на соответствующий вариант и формирует нужные лексемы, работая с метасимволами и переходя из одного состояния в другое.
- FSyntax - реализация разбора грамматик разбора, представленных по правилам YACC, методом прямого рекурсивного спуска.

Структуры данных

Структуры данных проекта в основном соответствуют понятиям, используемым при конструировании трансляторов. Большинство из них имеют методы печати у удобочитаемом виде. А основные процедуры сохранения и восстановления.

Лексемы

Лексема – структура (T_Lexeme), состоит из трех частей: тип (группа и собственно тип), место (позиция, длина, строка и колонка) и тело (значение и текст) для значимых лексем. В дальнейшем при подключении аппарата многофайлового текста в место добавится поле файл. За основу взяты лексемы в языках C и C++.

Тип лексемы состоит из группы и собственно типа. Группа лексемы в основном идентифицируют лексему и разбита на:

- None – пустая лексема, может определять инициализацию или быть результатом поиска;
- Oper – операция ,
- Separator – разделитель скобки, запятая, кавычки
- Integer - целое, значимая лексема у нее есть собственное тело, не выводимое из типа;
- Number - действительное, значимая лексема;
- String - строка, значимая лексема. В настоящее время в учебных языках нет строк, поэтому эти лексемы нигде не используются и сканером не распознаются;
- Ident - идентификатор, значимая лексема;

- Reserv – резервное слово. Для разных языков предполагается подключение различных таблиц таких слов; Определение типа **void**, **short**, **int**, **long**, **float**, **double** объединяются в **define**
- Space – пустые лексемы, пробелы комментарии а может быть и конец строки. Используются как любая лексема при разборе грамматики в Нормальной Форме Бекуса и при построении таблиц предпросмотра;
- Line – символ конца строки;
- Error – ошибочная лексема, выявляется сканером;
- Priority – объединение операций по приоритетам;
    - `Assign – =, +=, -=, *=, /=, ~=,`
    - `Question – ?`
    - `OrOr – ||`
    - `AndAnd – &&`
    - `Or – |`
    - `XOr – ^`
    - `And – &`
    - `Equ – ==,!=`
    - `Compare – >, >=, <, <=`
    - `Shift – >>, <<`
    - `Term – +, -`
    - `Factor – *, /, %`
    - `Unary – -, = !, ~`
    - `IncDec - --, ++`
- Syntax – групповая лексема распознанная по нескольким соседним лексемам. Может состоять из нескольких лексем, например, пустые скобки или доопределяться – идентификатор переменной, функции и массива. Реализуются классом A_Syntax
    - `IdentFunc – идентификатор функции`
    - `IdentArr – идентификатор массива`
    - `DefFunc – определение функции`
    - `DefArr – определение массива`
    - `DefVar – определение переменной`
    - `DefType – определение типа. Совокупность` **`static, const,`** `&` `и` **`define.`**
    - `Label - метка`
    - `Cast – приведение типов`
    - `CastUnary – приведение типов`
    - `EmptyCramp – пустые скобки {}`
    - `EmptyBracket – пустые скобки ()`
    - `EmptyScrape – пустые скобки []`
- Eof – конец исходной программы. Генерируются сканером, при выходе за пределы текста, без сдвига на следующую лексему.
- Preproc – лексемы препроцессора,
    - #ifdef и #ifndef – проверка на наличие определений макропеременной
    - #else
    - #endif
- Meta – метасимволы , используются при разборе грамматик разбора;
- Grammar- определяют символы вложенной грамматики.

Для значений значимых лексем (целое, число и идентификатор) в сканере создаются соответствующие таблицы значений, а в лексеме индекс строки в этих таблицах.

Представление лексем

В файлах описания грамматик, входных и выходных, в файлах таблиц разборщиков, в файлах представления грамматик в виде макрокоманд, лексемы представляются в следующем виде

- Произвольный идентификатор, соответствующий предварительно определенному токену, для файлов (*.YACC) и (*.y).
- Идентификатор формата группа_тип, где группа и тип соответствующие символьные представления группы и типа лексемы, для файлов (*.bnf), (*.bif) и (*.y).
- Группа символов вида <группа, тип> или <группа>, для файлов (*.y).
- Символьное представление лексемы, для файлов (*.bnf), (*.bif).
- Макрокоманды
    - `LEX(G), G – группа лексемы`
    - `SYM(S), S - разделитель`
    - `OPER(O), O - операция`
    - `WORD(W), W – ключевое слово`
    - `GROUP(X), X – тип лексемы` Syntax
    - `PRIOR(P), P - тип лексемы` Priority

Универсальный элемент GSymbol

Основные алгоритмы проекта настроены на обработку символов, поэтому в проекте используется универсальный символ GSymbol, имеющий следующие типы

- None – начальная инициализация, может задавать результат неудачного поиска;
- Formula – формула грамматики, индекс в таблице формул;
- Produce – продукция грамматики, индекс в формуле и ключ в таблице продукций;
- Terminal – терминальный символ, лексема без тела и без информации о месте в программе;
- Vector – последовательность элементов для реализации предпросмотров, групповых действий и таблиц LL(k), индекс в таблице векторов.
- Empty – пустой символ;
- Eof – символ конца файла исходной программы;
- Maker - действие, ключ в таблице действий;
- Point – пункт или точка в продукции;
- Para – двойной элемент, задающий место в продукции и дополнительный символ. Основа для реализации аппарата ситуаций или состояний;
- Situation – состояние анализатора, аналогичное состоянию конечного автомата;
- Table – таблица LL(k), индекс в таблице таблиц.

Вектор символов GSymbolVector

Реализует последовательность символов для элементов продукций, символов предпросмотров. В большинстве случаев это часть некой структуры. Имеется специальная таблица векторов, которая реализует групповые элементы.

Набор символов GSymbolSet

Реализует вектор символов. Для многих алгоритмов необходимо при добавлении элементов знать добавились ли они действительно, или там уже были. Для этого созданы операции += возвращающие соответствующее булевское значение.

Синтаксическая формула GFormula

Массив продукций или правил *GProduce*. Содержит заголовок *GSymbol head*, который ее идентифицирует.

Правило или продукция GProduce

Часть формулы *GFormula*, состоит из элементов *GSymbol*. Также содержит заголовок *GSymbol head* и действие *GSymbol maker*, куда помещаются действия при восходящем разборе.

Таблица формул GFormulaTable

Массив формул *GFormula*, является ядром класса TGrammar. Содержит основные методы сохранения и восстановления грамматики, печати и преобразования.

Класс TFirst

Основан на *GSymbolSet* и реализует FIRST для грамматики

Класс TFollow

Основан на *GSymbolSet* и реализует FOLLOW для грамматики

Класс TGrammar

Основная структура, описывающая грамматику, содержит таблицу формул, таблицы FIRST, FOLLOW и EFF, таблицу лексем, используемых в грамматике. Ссылки на таблицу действий, генератора таблиц разбора и сами эти таблицы.

Класс TFirstFollowMap

Таблицы FIRST, FOLLOW для грамматики имеют ключом универсальный символ *Таблицу* или *Терминал*, а элементом соответственно FIRST или FOLLOW.

Класс TTableKey

В алгоритмах проекта используются таблицы с ключами. В качестве ключа часто выступает пара универсальных символов *GSymbol key*, *sym*. Причем в качестве первой части ключа часто выступает *Formula*, а второй *Terminal*.

Класс TTableMap

Данный класс используется, если элементом таблицы выступает универсальный символ. К таким таблицам относится таблица нисходящего распознавателя и таблица переходов для восходящего разбора.

Структура TAction

Основной элемент таблицы восходящего разбора структура TAction.

- None – начальная инициализация, может задавать результат неудачного поиска;
- Shift – сдвиг
- Reduce – свертка
- Accept – успех
- Goto – переход
- Error – ошибка
- Fi – фи недостижимость.

Класс LRTableMap

Основная таблица восходящего разбора. Ключ TTableKey элемент TAction.

Генерация таблиц разбора

Генерируются таблицы как для нисходящего разбора LL, так и для восходящего (LR, LALR). Кроме самих таблиц разбора разными для частного и общего случая могут быть и вспомогательные таблицы.

От 0 и 1 к k.

Таблицы соответствуют как для частных случаев LL1, LR0, LR1 и LALR, так и для общих LLk, SLRk, LRk и LALRk. Для частных случаев имеются оптимизированные алгоритмы построения таблиц разбора, поэтому они представлены наряду с общими случаями. Они проще для понимания, отдельно описаны в литературе, да и были реализованы первыми. Могут рассматриваться как проверка общих алгоритмов при k = 1.

Для реализации предпросмотров добавляется аппарат цепочек символов, таблица векторов. И вводится новый тип терминального символа Vector, который указывает на строку данной таблицы. Эта таблица добавляется к таблице грамматики и используется как при построении таблиц разбора, так и при самом разборе табличным разборщиком GSyntax.

Нисходящий разбор

Нисходящий разбор имеет меньше вариантов построения, он реализован для вариантов таблиц LL1 и LLk. Сам разбор может отличаться действиями, выполняемыми при разборе и вариантами использования стека разбора.

Восходящий разбор

Вначале строится таблица ситуаций или состояний и таблица переходов. Затем по ним строится таблица действий.

Таблицы ситуаций могут быть трех видов

- SLRk – простые таблицы
- LRk – канонические таблицы
- LALRk - с символами предпросмотра. Таблицы ситуаций строятся либо на основе канонических таблиц, либо на основе простых таблиц, с использованием аппарата распространения символов, либо с использованием `базисных пунктов ядер`.

Кроме того, таблицы восходящего разбора для всех трех видов могут строиться на основе двух технологий

- `CLOSURE` - аппарата замыкания,
- `CORE` - `базисных пунктов ядер. Класс TC2A.`

Класс A_LRBuilder

Для построения таблиц восходящего разбора для общего случая с k символами предпросмотра используется специальный класс A_LRBuilder. Перекрытием методов, которого, определяется вид генерируемой таблицы. Методы разбиты на две группы:

- Построение таблиц состояний и переходов, метод `make_States()`.
- Построение таблицы разбора, метод `make_Table()`.

	A_LR	LR0	LR1	SLRk	LR0Core	LRk	LRkCore	LALRkCore	Splitting
Init		::	::	Init3	Init	Init3			::
FirstTailSym	::								::
make_States	::							::	
make_Action	::								
Closure		::	::	BetaFirstKClosure	{}	BetaFirstKClosure	{}	LRkCore	BetaFirstKCl
ActionShift		SymShift	SymShift	::	::	EFFShift	EFFShift	LRkCore	::
ActionReduce		FollowReduce	ParaSymReduce	FollowReduce	FollowReduce	ParaSymReduce	ParaSymReduce	LRkCore	::
CLOSURE	::								
GOTO	::				::		::	LRkCore	
MuFirstTailSym	-	-		-	::	-	::	LRkCore	

Программы

TranParse – лексический разбор.

Тестирование модулей лексического анализа. Преобразование исходного текста программы в массив лексем выполняют несколько модулей.

Модуль Scan читает собственно исходный текст либо их файла, либо из строки.

Модуль Parse пропускает пробелы и комментарии, организует механизм возврата. Работает в режиме Неуправляемого сканера. В этом модуле реализована условная компиляция. В дальнейшем в модуле будут реализован аппарат макрокоманд и вставок файлов исходного текста.

Модуль Lexer организует связь со сканером, сгенерированным программой Flex для грамматик C и C99.

Модуль ASyntax формирует групповые лексемы в режиме Управляемого сканера.

Вход тестируемая программа, выход:

- Печать исходного текста
- Печать лексем.

Сама по себе программа не используется, ее модули применяются при чтении описателя грамматики и текста исходной программы.

TranSyntax – написанные вручную программы нисходящего разбора

Программы написаны вручную, с учетом грамматики учебного языка. По своей структуре они практически повторяют структуру грамматики, это хорошо видно если сравнить их, и особенно MSyntax, со сгенерированными текстами (PSyntax) и соответствующим описанием грамматики.

- Syntax - Проверка синтаксиса, выход Да/Нет. Совокупность функций по разбору конструкций языка, возвращающих информацию о их правильности. Ни каких дополнительных действий. Работа до первой ошибки.
- TSyntax – Построение синтаксического дерева, выход нетипизированное дерево. Как и в первом случае это набор аналогичных функций. Результатом каждой функций является соответствующий узел в дереве.
- CSyntax – Генерация кода стековой машины, выход - коды стековой машины. С помощью функций модуля семантики непосредственно в ходе разбора строятся стеки блоков и описаний разбираемой программы. На основании этой информации определяются типы переменных и результатов операций, что позволяет генерировать коды стековой машины.
- MSyntax – Построение дерева и генерация кода с помощью Действий. Выполняется работа аналогичная предыдущим двум пунктам, только делается это с помощью аппарата Действий.

Результаты работы этих программ служат в дальнейшем, в тестах, эталоном при сравнении.

TranYACC – генерируемые вставки на C++

Вставки бывают трех видов:

- Выходные файлы системы генерации компиляторов YACC (YSyntax). Соответственно входные файлы для YACC могут быть:
    - созданы вручную, расширение *.y.
    - сгенерированы TableBuilder (режим YACC) на основе грамматик (LL, LR, NR).
- Тексты программ прямого рекурсивного спуска (PSyntax), сгенерированные Table Builder (режим PROG) на основе грамматик (LL, NL).
- Тексты формирующие таблицы разбора, сгенерированные TableBuilder (режим Table).

После генерации вставок необходим этап трансляции TranYACC с помощью C++. Результатом работы кроме дополнительной информации и сообщении о правильности программы являются:

- Построение синтаксического дерева и/или генерация кодов стековой машины. (Tree, Code)
- Создание дерева узлов. (Node)

TranRules – нисходящий разбор с использованием грамматик (LL, NL).

Распознаватель работает по методу рекурсивного спуска создавая рекурсию либо вызовом функций, либо используя соответствующий стек. Генерация кода и построение синтаксического дерева посредством Действий. При возвратах восстанавливается только состояние сканера, внутренние таблицы и коды не отбрасываются. Считается, что возврат возникает до начала действий. Для интерпретации используются таблицы грамматик (LL, NL) и читаемые из файла.

- Formula и Produce – рекурсивные функции работающие на основе формул грамматики;
- FormulaStack – использование стека, для запоминания места разбора, формул и продукций. Элемент стека можно рассматривать как некий курсор, указывающий на применяемое в данный момент правило грамматики;
- RuleStack – раскрытие тела продукции в стек.
    - `ParseStack_SN()` - дерево собирается в основном стеке разбора,
    - `ParseStack()` - дерево собирается в отдельном стеке LexNodeStack.

  При выборе продукции для раскрытия в стеке, может быть применен аппарат, аналогичный используемому при построении таблицы нисходящего разбора.

TranGrammar – табличные распознаватели.

Работают с двумя видами таблиц, для нисходящего и восходящего разбора.

- LL – нисходящий разбор, грамматика LL, вывод –
    - синтаксическое дерево (Tree),
    - семантическое дерево (Tree*),
    - код стековой машины (Code).
    - автоматически генерируемое синтаксическое дерево (Auto).

  Генерация таблицы – LL1, LL(k).

- LR – восходящий разбор, грамматики (LL, LR, NR), вывод –
    - синтаксическое дерево (Tree),

- семантическое дерево (Tree*),
- код стековой машины (Code),
- дерево узлов (Node),
- автоматически генерируемое синтаксическое дерево (Auto).

Генерация таблицы –

- LR0 (LR0, Core), LR1 (LR1, Core), LALR (LALR, Full, Tab, Core),
- SLR(k), LR(k) (LR, Core), LALR(k) (LALR, Full, Tab, Core), k >= 1.

У программы два входных файла: таблица разбора – выход программы TableBuilder и текст транслируемой программы. Результатом работы кроме дополнительной информации и сообщении о правильности программы являются:

- Построение синтаксического дерева в дополнительном стеке и/или генерация кодов стековой машины. (Tree, Code).
- Создание дерева узлов в основном стеке (Node) или дополнительном стеке (Turn).
- Построение синтаксического дерева в основном стеке (Made).
- Построение семантического дерева(Tree*), в режиме двухфазной работы.
- Автоматически генерируемое синтаксическое дерево (Auto).

Разбор входного текста и построение дерева или кодов стековой машины выполняется одним из восьми методов.

- ParseLL1() – нисходящий разбор без возвратов с созданием дерева (Tree) и кодов стековой машины (Code). Промежуточная информация хранится во внешних структурах класса MC_Syntax.
- ParseLL1_SN() - создание дерева узлов (Node). Промежуточная информация хранится в самом стеке разбора.
- ParseLLk () – тоже что и в методе ParseLL1(), но с использованием предпросмотра глубиной k символов, создание автоматического дерева узлов (Auto).
- ParseLLk_SN() – тоже что и в методе ParseLL1_SN (), но с использованием предпросмотра глубиной k символов, создание дерева узлов (Node). Промежуточная информация хранится в самом стеке разбора, создание автоматического дерева узлов (Auto).
- ParseLR0() – тоже что и в методе ParseLL1(), но с использованием восходящего разбора, создание автоматического дерева узлов (Auto).
- ParseLR0_SN() – тоже что и в методе ParseLL1_SN (), но с использованием восходящего разбора, создание автоматического дерева узлов (Auto).
- ParseLRk()– тоже что и в методе ParseLR0 (), но с использованием предпросмотра глубиной k символов и построением автоматического дерева узлов (Auto).
- ParseLRk_SN()– тоже что и в методе ParseLL1_SN (), но с использованием восходящего разбора, создание автоматического дерева узлов (Auto).

С помощью модуля CGrammar реализуется синтаксический разбор для грамматик C и C99.

TableBuilder - генератор таблиц разбора и программных вставок

Входными данными является грамматика, заданная одним из следующих способов:

- описание грамматики (LL, NL, LR, NR, …) в формате MACRO в теле программы, режимы (0x00, … , 0x09);
- входной файл в формате системы YACC, режимы разбора (0x0E, 0x0D, 0x0C). Первый режим - это прямой рекурсивный спуск с действиями в виде программных вставок, второй реализован с использованием таблицы восходящего метода, третий использование программной вставки, сгенерированной YACC;
- запись грамматики в Нормальной Форме Бекуса режим разбора (0x0B). Метод BNF (сейчас не используемый) предполагает, что строится автоматическое дерево разбора, которое потом преобразуется в дерево для режима (0x0D);
- запись грамматики в Итерационной Форме Бекуса режим разбора (0x0A).
- файл грамматики в формате (*.tab) выход программы TableBuilder режим (0x0F).

Выход программы:

- таблицы разбора, в зависимости от метода (LL, LR, LALR), и таблицы грамматики;
- входной файл для **YACC** или НФБ
- программа разбора методом прямого рекурсивного спуска (*.cpp, *.h), сгенерированная методом PROG.

Кроме того, может быть выдано много служебной информации:

- Grammar = 0x00000001, GrammarItem = 0x00000002, грамматика после преобразований
- Formula = 0x00000004, FormulaItem = 0x00000008, грамматика до преобразований
- Maker = 0x00000010, MakerItem = 0x00000020, перечень Действий
- Lexeme = 0x00000040, LexemeItem = 0x00000080, таблица лексем, используемых в грамматике
- First = 0x00000100, FirstItem = 0x00000200, список лексем, которыми могут начинаться терминальные и нетерминальные символы грамматики
- Follow = 0x00000400, FollowItem = 0x00000800, список лексем, которые могут следовать за нетерминальными символами грамматики
- Action = 0x00001000, ActionItem = 0x00002000, таблица для варианта восходящего разбора LR
- Cell = opAction, CellItem = ActionItem таблица разбора для варианта нисходящего спуска LL
- State = 0x00010000, StateItem = 0x00020000, StateCore = 0x00040000, StateSort = 0x00080000, варианты выдачи таблицы состояний, ситуаций для метода восходящего разбора LR
- GoTo = 0x00400000, GoToItem = 0x00800000, таблица переходов из состояния в состояние
- Rule = 0x00100000, RuleItem = 0x00200000, таблица правил
- Ahead = opC2A, AheadItem = C2AItem, служебная таблица для LALR
- C2A = 0x00004000, opC2AItem = 0x00008000, служебная таблица для LALR_Core

Этапы создания и выполнения.

Программы отличаются друг от друга количеством этапов создания и выполнения программы. Написанные вручную TranParse и TranSyntax оттранслированы и выполняются за один этап.

TranRules состоит из двух частей: функций разбора и описаний грамматик, заданных макрокомандами, оттранслирована и выполняется за один этап.

TableBuilder выступает в качестве первого этапа для многоэтапной работы. Возможны варианты работы:

- Генерация таблиц, TableBuilder –> таблица разбора -> TranGrammar
- Генератор YACC , TableBuilder –> *.y -> YACC -> *.tab.c –> C++ -> TranYACC
- Генератор функций разбора, TableBuilder –> функций разбора -> C++ -> TranYACC

Дополнительная информация, выдаваемая на этапе разбора.

При работе программ разбора может быть выдана дополнительная информация о ходе разбора

- 0x0001 - other, 0x0002 - space, 0x0004 - line, печать входных лексем. При возможных возвратах лексемы будут печататься каждый раз при их запросах.
- 0x0008 - Lexeme, печать лексемы для варианта Управляемого сканера, когда разбор идет по таблицам и без возвратов.
- 0x0010 - print_syntax_tree, печать синтаксического дерева без типов
- 0x0020 - erase_code, включение оптимизации таблицы кодов
- 0x0040 - print_code, печать команд стековой машины по ходу их генерации
- 0x0080 - print_operator,
- 0x0100 - print_rule, печать правил грамматики в ходе разбора
- 0x0200 - print_maker, печать Действий в ходе разбора
- 0x0400 - print_stack, печать стека
- 0x0800 - print_TableCode, печать таблицы кодов стековой машины
- 0x1000 - print_SNode печать промежуточного дерева узлов
- 0x2000 - print_semantic_tree печать семантического дерева с типами и определением идентификаторов.

Фазы работы программ

Разбиение процесса трансляции на фазы общеизвестно. В проекте это используется для показа возможностей работы с семантической информацией.

Под кодогенерацией понимается все что за синтаксисом и семантикой.

Трехфазная работа

При трехфазной работе все действия выполняются раздельно и могут комбинироваться все доступные режимы

<table>
<tr><td colspan="3">Первая фаза</td><td>Вторая фаза</td><td>Третья фаза</td></tr>
<tr><td colspan="3">Синтаксис</td><td>Семантика</td><td>Кодогенерация</td></tr>
<tr><td>1</td><td colspan="2">PSyntax (Tree)</td><td rowspan="11">BlockTable, TreeSearch</td><td rowspan="11">CodeGenerate, QuadGenerate</td></tr>
<tr><td>2</td><td colspan="2">YACC (Tree, Made)</td></tr>
<tr><td>3</td><td colspan="2">GSyntax (Tree, Made)</td></tr>
<tr><td>4</td><td colspan="2">RSyntax (Tree, Made)</td></tr>
<tr><td>5</td><td>YACC(Node)</td><td rowspan="3">Create</td></tr>
<tr><td>6</td><td>RSyntax (Node)</td></tr>
<tr><td>7</td><td>GSyntax (Node)</td></tr>
<tr><td>8</td><td>GSyntax (Turn)</td><td>Make</td></tr>
<tr><td>9</td><td colspan="2">MSyntax(Tree)</td></tr>
<tr><td>10</td><td colspan="2">TSyntax*</td></tr>
</table>

Результатом работы первой фазы является синтаксическое дерево. Либо сразу для режимов Tree и Made, макрокоманды MAKE_TREE модуля MakeTree.cpp. Либо через промежуточное дерево универсальных узлов для режимов Node и Turn, методы make_yacc_XX_YY модуля MakeYACC.cpp.

*- TSyntax строит дерево непосредственно, без использования Действий, функции, разбирающие конкретную формулу грамматики, возвращают построенный узел дерева.

Во время второй фазы осуществляется обход синтаксического дерева. Либо с построением таблиц блоков и идентификаторов, с получением из них семантической информации - BlockTable. Либо семантическая информация получается непосредственно из дерева - TreeSearch.

На третьей фазе при обходе семантического дерева строятся команды стековой машины или четверки.

Двухфазная работа

При двухфазной работе объединены синтаксис и семантика, семантическое дерево строится сразу, минуя синтаксическое.

<table>
<tr><td colspan="2">Первая фаза</td><td>Вторая фаза</td></tr>
<tr><td colspan="2">Синтаксис</td><td>Семантика</td><td>Кодогенерация</td></tr>
<tr><td>1</td><td>RSyntax (Made)</td><td rowspan="3">BlockTable, StackSearch (rule_stack)</td><td rowspan="8">CodeGenerate, QuadGenerate</td></tr>
<tr><td>2</td><td>GSyntax (Made)</td></tr>
<tr><td>3</td><td>YACC (Made)</td></tr>
<tr><td>4</td><td>PSyntax (Tree)</td><td rowspan="5">BlockTable, StackSearch (tree_stack)</td></tr>
<tr><td>5</td><td>RSyntax (Tree)</td></tr>
<tr><td>6</td><td>YACC (Tree)</td></tr>
<tr><td>7</td><td>GSyntax (Tree)</td></tr>
<tr><td>8</td><td>MSyntax(Tree)</td></tr>
</table>

При двухфазной работе возможны режимы Tree и Made, строящие синтаксическое дерево непосредственно, минуя дерево универсальных узлов. Вариант получения семантической информации из стека использует соответствующий стек, в котором находятся ветви собираемого дерева.

Однофазная работа

При однофазной работе синтаксическое и семантическое деревья не строятся, строится только таблицы блоков и идентификаторов - BlockTable. Семантическая информация берется из этих таблиц.

<table>
<tr><td colspan="3">Первая фаза</td></tr>
<tr><td colspan="2">Синтаксис</td><td>Семантика</td><td>Кодогенерация</td></tr>
<tr><td>1</td><td>PSyntax (Code)</td><td rowspan="6">BlockTable</td><td rowspan="6">CodeGenerate</td></tr>
<tr><td>2</td><td>RSyntax (Code)</td></tr>
<tr><td>3</td><td>GSyntax (Code)</td></tr>
<tr><td>4</td><td>MSyntax (Code)</td></tr>
<tr><td>5</td><td>YACC (Code)</td></tr>
<tr><td>6</td><td>CSyntax</td></tr>
</table>

Возможна генерация только команд стековой машины, так как это выполняется действиями, а не фазой обхода семантического дерева. CSyntax генерирует команды стековой машины и работает с таблицами блоков и идентификаторов непосредственно, без использования Действий.

Варианты создания узла дерева.

Внутреннее представление программы.

Структуры данных

В ходе разбора программа представляется в нескольких видах.

- Исходный текст (Source) - текстовый файл или строка.
- Лексемы (Lexemes) – структуры, состоящие из трех частей: тип, место и тело. Лексемы, соответствующие операциям, числам, идентификаторам, ключевым словам, определяющим тип, являются значимыми, так как в них содержится семантическая информация.
- Групповые лексемы (Syntax), могут *содержать* несколько лексем, а могут только создаваться после просмотра нескольких лексем. Групповые лексемы за исключением лексем пустых скобок – значимые лексемы.
- Синтаксическое дерево (Tree). Создается на этапе синтаксического разбора программы, каждый узел соответствует языковой конструкции в программе и большинство из узлов содержит лексему исходной программы. Типы языковых конструкций определяют класс узла. У каждого есть метод добавления семантической информации в дерево (Variables). В результате работы этого метода в дереве формируются ссылки на типы и определения переменных, и дерево преобразуется в семантическое.
- Семантическое дерево (Tree*). У каждого узла есть метод (Codes), с помощь которого, генерируются промежуточное представление целевой программы.
- Промежуточное представление целевой программы создается:
    - при синтаксическом разборе, либо непосредственно в методах класса CSyntax, либо в действиях MAKE_CODE;
    - при просмотре семантического дерева, метод (Codes).

  Есть два вида промежуточного представления:
    - Коды стековой машины (Codes).
    - Четверки. Метод `IsQuad()`.
- Дерево узлов (Nodes). Промежуточная структура данных, также соответствующая синтаксическим конструкциям транслируемой программы. Основой информацией которой являются лексемы и иерархия программы. Дерево состоит из двух типов узлов, простой узел SNode и контейнер SNodeList. Типы языковых конструкций определяют значение специальной переменной в этих узлах.

  Дерево узлов может быть построено двумя видами действий:
    - NODE совместно с YACC строят данное дерево, а затем с помощью метода (Create) строится синтаксическое дерево (Tree), пригодное для дальнейшей обработки.
    - TURN - данное дерево создается в дополнительном стеке, а затем с помощью метода (Make) строится синтаксическое дерево (Tree).
- Автоматически генерируемое синтаксическое дерево (Auto). Состоящее из узлов двух типов – Лексема (содержит лексему) и Продукция (содержит номер продукции). Это дерево полностью соответствует описанию грамматики. Есть возможность упростить это дерево, выбрасывая цепные ссылки.

Синтаксическое дерево

При создании дерева, на этапе синтаксического разбора, оно создается узел за узлом, снизу-вверх. Это вызвано тем, что нижестоящий узел не может себя вставить в вышестоящий. За исключением построения `AutoTree для нисходящего разбора`. При таком построении необходимо иметь место для

хранения промежуточных результатов сборки. Это может быть выделенный стек узлов дерева, работающий в паре со стеком значимых лексем - LexNodeStack, или сам стек разбора - ParseStack.

Семантическая информация.

Семантическая информация необходима для непосредственного построения кодов целевой машины, при разборе программы - Code. Или при построении семантического дерева - Tree*. Эта информация может храниться в таблице блоков и идентификаторов - BlockIdent, или в синтаксическом дереве – Tree, или в стеке построения дерева или в стеке разбора Stack.

Таблица блоков и идентификаторов.

Строится либо при разборе программы - Lex, либо при обходе синтаксического дерева - Tree.

Можно выделить три случая использования этой таблицы:

- Lex -> BlockIdent -> Code – построение кодов целевой машины непосредственно при просмотре исходного текста программы.
- Lex -> BlockIdent -> Tree* – построение семантического дерева непосредственно при просмотре исходного текста программы.
- Tree -> BlockIdent -> Tree* - построение семантического дерева при обходе синтаксического дерева на дополнительном проходе.

Получение семантической информации из синтаксического дерева.

Синтаксическое дерево может быть полностью построенным, тогда информация получается при его просмотре – TreeSearch, или частично построенным, тогда информация хранится в стеках LexNodeStack(tree_stack) или ParseStack(rule_stack).

Можно выделить три случая использования этой информации:

- Tree -> TreeSearch -> Tree* - построение семантического дерева при обходе синтаксического дерева с поиском по дереву.
- Lex -> LexNodeStack -> Tree* - построение семантического дерева на этапе разбора программы с использованием дополнительных стеков.
- Lex -> ParseStack -> Tree* - построение семантического дерева на этапе разбора программы с использованием стека разбора.

Получение семантической информации из стека.

Два последних случая получают информация из частично построенных деревьев, с другой стороны части дерева лежат в соответствующем стеке и можно считать, что информация берется из стека.

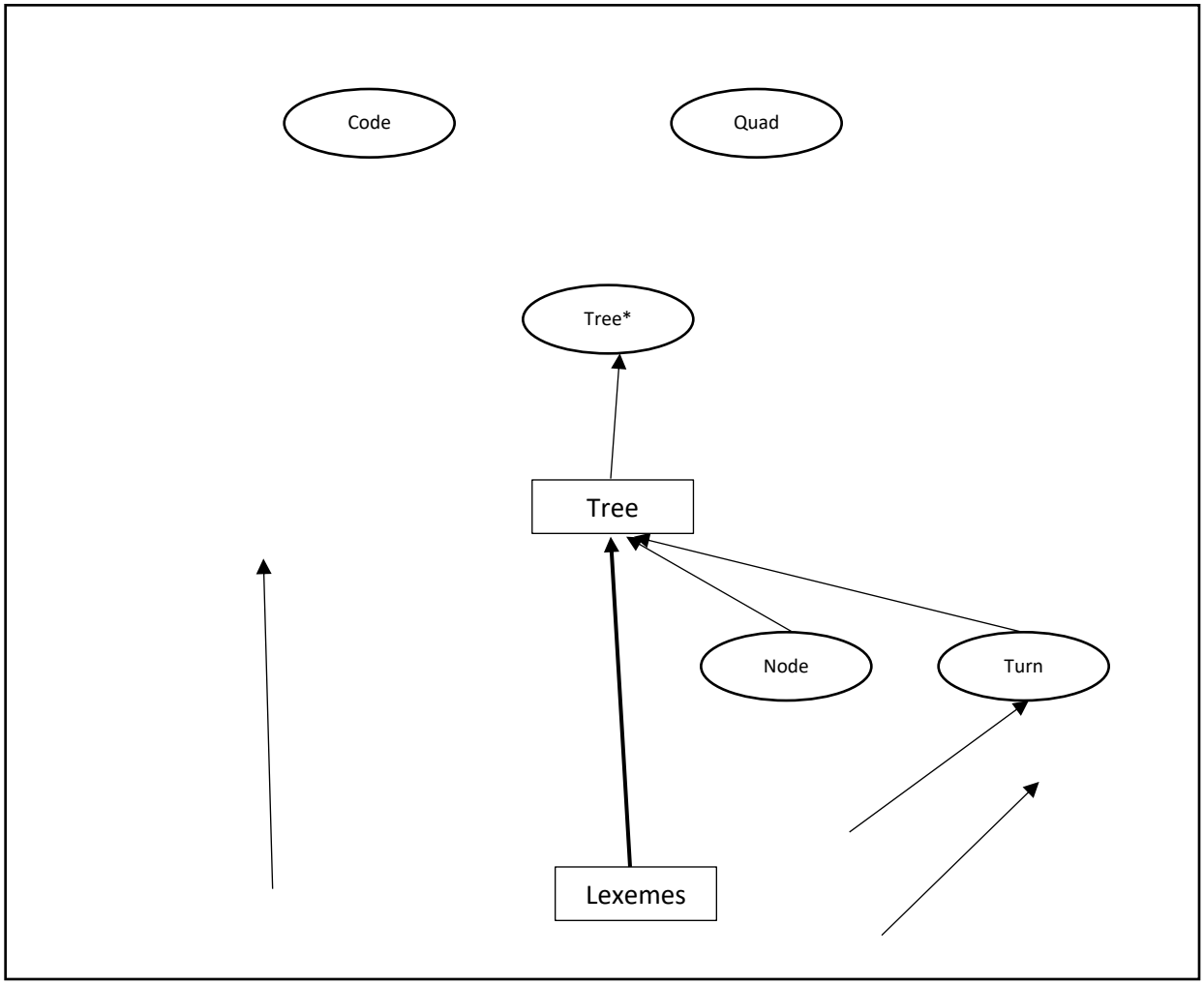
Стек разбора – **rule_stack**, режим - **Made** и стек построения синтаксического дерева **tree_stack** режим - **Tree**. В этих стеках находятся левые части собираемого дерева. В них в свою очередь находится информация о наследуемых атрибутах.

Семантическое дерево

При построении семантическое дерево, преобразуется из синтаксического. Структура дерева практически не меняется, добавляются новые ссылки между узлами, записывается информация о типах, формируются узлы преобразования типов, если это необходимо.

Есть несколько способов построения семантического дерева.

- С предварительным построением синтаксического дерева и дальнейшим его обходом. Информация из дерева. Есть два способа получения информации из синтаксического дерева:
    - Построение, при обходе дерева, таблицы блоков и таблицы идентификаторов, которые представляют собой стеки, содержание которых соответствует точкам синтаксического дерева. Метод IsBlock();
    - Поиск информации о типах и переменных выполняется просмотром дерева, без вспомогательных таблиц. Метод IsTree().
- Построение семантического дерева напрямую на этапе синтаксиса. В этом случае узлы, частично построенного дерева, хранятся в стеке разбора. Информация - наследуемые атрибуты - получаются путем просмотра стека разбора. Метод IsStack ().



Целевая программа

Четверки.

Промежуточное представление целевой программы в виде четверок, трехадресных команд.

Команды стековой машины.

Представление целевой программы в виде команд некой стековой машины. Реализации этой стековой машины нет.

T_Command

- Cast – приведение типов
- Oper - операция
- Call – вызов функции
- GoTo - переход
- Label - метка
- NewVar – создание в стеке переменной
- NewArr – создание в стеке массива
- PushVar - адрес переменной на вершину стека
- PushVal - значение переменной на вершину стека
- PushArr - адрес индексируемой переменной на вершину стека
- PushAal - значение индексируемой переменной на вершину стека
- PushInt – целое на вершину стека
- PushNum – число на вершину стека
- Pop – сброс вершины стека
- Return – возврат значения
- Exit- выход из функции
- InitArr – инициализация массива

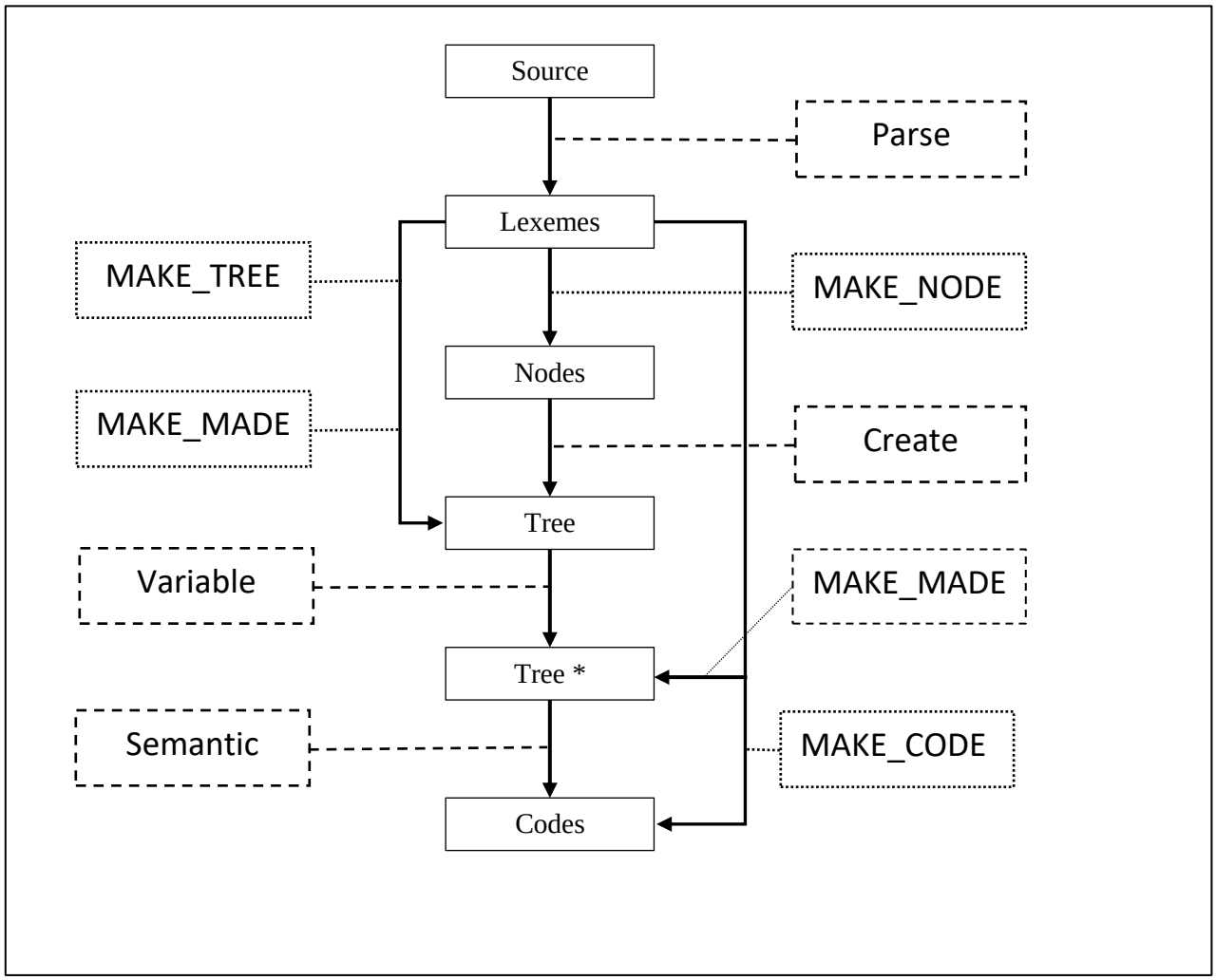
Стековая машина

Стековая машина состоит из Стека, в котором будут храниться данные программы обрабатываемой программы. Массива команд по обработке этих данных и Вычислителя, выполняющего эти команды.

Команды машины делятся на несколько типов:

- Команды работы со стеком
- Управляющие команды
- Операции

Преобразование программы



Этапы трансляции, результат этапа и способ использования семантической информации на данном этапе.

- Лексический разбор
    - Массив лексем
- Синтаксис
    - Синтаксическое дерево
    - Семантическое дерево
        - Таблица блоков и идентификаторов. Lex -> BlockIdent -> Tree*
        - Стеки лексем и узлов дерева. Lex -> LexNodeStack -> Tree*
        - Стек разбора. Lex -> ParseStack -> Tree*
    - Команды стековой машины
        - Таблица блоков и идентификаторов. Lex -> BlockIdent -> Code
- Семантика
    - Семантическое дерево
        - Таблица блоков и идентификаторов. Tree -> BlockIdent -> Tree*
        - Поиск по дереву. Tree -> TreeSearch -> Tree*
- Генерация промежуточного кода
    - Команды стековой машины
    - Четверки
- Оптимизация

Действия

При синтаксическом разборе могут быть выполнены некоторые действия. Эти действия различаются по тому что они делают и по тому как они реализованы.

Сам синтаксис отвечает только на вопрос Да/Нет относительно правильности программы. Для того, чтобы представить программу в некотором внутреннем виде, необходимо выполнить некие действия.

Действия могут задаваться текстом C++ вставки в программу синтаксического разбора (TSyntax, CSyntax), или указанием заранее определенного действия с помощью макрокоманды в программе синтаксического разбора (MSyntax) или элементом Действие в описании грамматики. Последнее, в сгенерированных синтаксических разборщиках (YSyntax, PSyntax), в дальнейшем преобразуется во вставку вызова макрокоманды.

Действия - в конечном счете это функции на C++. Они делятся по параметрам на группы

- Tree – работа со структурами по формированию синтаксического дерева, оно формируется во вспомогательном стеке, в отдельном стеке запоминаются значимые обрабатываемые лексемы. Макрокоманда - MAKE_TREE;
- Code – работа со структурами по непосредственной генерации кодов стековой машины; Макрокоманда - MAKE_CODE;
- Node – доступ к стеку разбора, работа с деревом узлов (Nodes), затем метод Create формирует дерево (Tree). Макрокоманды - MAKE_NODE, MAKE_YACC;
- Made – доступ к стеку разбора, непосредственное формирование в этом стеке дерева (Tree). Задание действий в грамматике совпадает с Node;
- Turn – создание во вспомогательном стеке дерева узлов (Nodes), с использование только узлов контейнеров, затем метод Make формирует дерево (Tree). Действия имеют вид универсальных команд по созданию дерева. Макрокоманда - MAKE_TURN.

Есть объединяющие макрокоманды: MAKE (MAKE_TREE, MAKE_CODE), MAKE_ALL (MAKE_TREE, MAKE_CODE, MAKE_YACC).

Заголовки функций формируются соответствующими макрокомандами, в которых указывается Имя действия – идентификатор, с которым связывается индекс в таблице действий.

Действия работают с данными внутреннего представления программы:

- Стек узлов синтаксического дерева
- Стек лексем
- Таблица блоков и идентификаторов
- Таблица объявленных функций
- Таблица операций и приведений
- Стек типов
- Стек разбора и узлов дерева.

При работе действий Tree, Code и Turn значимые лексемы пишутся разборщиком в стек лексем и должны оттуда вычитываться действиями.

При работе действия под управлением синтаксического разбора, допускающего возвраты, может потребоваться необходимость удаления созданных узлов и запомненных лексем. Это реализуется аппаратом соответствующих стеков. В режиме генерации кодов возврат не возможен.

В большинстве случаев перед началом работы с действиями в продукциях остаются только действия одной группы - (Tree, Code), (Node), (Turn).

Tree – формирование синтаксического дерева.

Действия Tree формируют дерево типизированных синтаксических узлов в стеке узлов. В конце успешной работы в стеке сформирован корневой узел. Используется стек лексем, для сохранения значимых.

Code – генерация команд.

Эти действия работают при однофазном режиме работы. Поэтому кроме собственно формирования команд, необходима еще работа по созданию семантической информации. Что и делается созданием Таблиц блоков и Идентификаторов.

Node и Made – использование стека разбора.

При работе действий Node и Made узлы дерева, при сборке, хранятся непосредственно в стеке разбора, там же и хранятся читаемые лексемы. Положение в стеке соответствует положению элементов в продукции. В действиях при ссылках необходимо учитывать это положение. Поэтому эти действия не могут быть использованы в грамматиках, записанных по правилам BIF, в которых производится преобразование продукций с их объединением и расщеплением.

Ссылки на элементы стека задаются списком номеров, порядок присваивания номеров отличается для нисходящего и восходящего разборов.

Для восходящего разбора номер элемента в стеке соответствует номеру элемента в продукции, с учетом фильтра Действий по группам. Так как перед работой действия сборка всех элементов слева от него уже произведена. Нулевым элементом считается результат самой продукции. Если действий в продукции нет, то выполняется действие по умолчанию – перенос информации первого элемента в нулевой.

Для нисходящего разбора элементы слева от действия уже разобраны и удалены из стека, а элементы справа еще не разобраны и в стеке для них нет информации, а только заготовлено место. Правее последнего элемента отведено место под результат продукции в целом. При завершении разбора продукции, информация, сформированная ее действиями, передается на верхний уровень. Таким образом при нисходящем разборе доступны только два элемента в стеке, не считая пустых элементов справа, само действие с номером равным 1 и формируемый результат, с номером на единицу больше номера самого правого из оставшихся элементов.

Turn – команды по созданию дерева.

Действия Turn формируют дерево не типизированных универсальных узлов в соответствующем стеке. Результат – единственный корневой узел.

Данные действия работают со стеком узлов и стеком лексем. В стек лексем разборщиком записываются значимые лексемы, которые могут служить источником семантической информации.

- None – копирование собранного дерева.
- Push – создание на вершине стека узла с лексемой с вершины стека лексем.

- Put – создание узла на вершине стека с пустой лексемой.
- Add – создание узла с лексемой и добавление его в узел на вершине стека.
- Make – создание узла с лексемой, вставка в него узлов с вершины стека и запись в стек.
- Down – чтение узла с вершины и добавление в узел на вершине.
- Up – к узлу на вершине добавляется узел под ним.
- Swap – два верхних узла меняются местами.
- Command – выполнение команд, заданных вторым параметром.
    - Lexeme – в зависимости от параметра
        - 0 – удаление лексемы с вершины стека
        - = -n – удаление с вершины стека n лексем.
        - = 1 – Запись лексемы в узел на вершине стека
    - Param – запись параметра узлу на вершине.

Для варианта Turn нет необходимости писать действия как функции C++, все указанные команды реализованы и могут быть использованы. Правда потом необходимо на основании полученного дерева не типизированных узлов, создавать дерево типизированных узлов. В проекте есть два варианта таких узлов, узлы синтаксического дерева учебного языка и таблица формул грамматики, при работе разборщика грамматики.

Сгруппированные действия.

При восходящем разборе, для действия, если оно не последний элемент продукции, создается временная формула с одной пустой продукцией, в заголовок которой переносится действие. Для действий типа Turn, которые задают элементарные действия над стеком узлов, может понадобиться последовательность действий, что в свою очередь приведет к набору временных формул. Поэтому есть возможность сгруппировать соседние действия в одно, представив его вектором. Группировать можно действия, в которых нет привязки к месту в продукции. Это действия типа: Tree, Code и Turn.

Тесты

В проекте представлены несколько грамматик учебные (ETF и другие), грамматики учебного языка (LL, NL, LR, NR) и грамматики C и C++. Есть несколько способов реализации синтаксических анализаторов, различные методы построения табличных анализаторов, несколько способов записи и разбора грамматик. Комбинация всего этого позволяет построить расширенную систему тестов, показывающую разнообразие аспектов построения компиляторов.

Parse

- TranParse.exe – `help`, выдача параметров
- TranParse.exe 0 0007 - `print_source, посимвольная печать входного файла`
- TranParse.exe 1 0001 - scan_source, просмотр лексем
- TranParse.exe 2 0001 - scan_next
- TranParse.exe 3 0001 - scan_control, управляемый сканер
- TranParse.exe 4 0008 - scan_lexeme, простые лексемы
- TranParse.exe 5 0008 - scan_lexeme, лексемы приоритетов
- TranParse.exe 6 0008 test_group.cpp - scan_lexeme, групповые лексемы
- TranParse.exe 7 0007 - `scan_store, тестирование откатов`
- TranParse.exe 8 4001 zetf.cpp - `etf_parse, разбор простого выражения`
- TranParse.exe 9 0008 - scan_write, запись лексемы в файл
- TranParse.exe A 0007 test_lexer.cpp - scan_lexer, сканер сгенерирован для C_99

Syntax

- :SYNTAX – только проверка синтаксиса.
- :TSYNTAX – кроме проверки построение синтаксического дерева, затем семантического, а затем генерация кола.
- :CSYNTAX – непосредственное построение кода. Первоисточник для большинства проверок генерации кода.
- :MSYNTAX_TREE – тоже что и :TSYNTAX, дерево и коды создаются действиями
- :MSYNTAX_CODE – тоже что и :CSYNTAX

Rules

:RULE_STACK - раскрытие тела продукции в стек.

:FORMULA_STACK - использование стека, для запоминания места разбора, формул и продукций.

:FORMULA_PRODUCE - рекурсивные функции работающие на основе формул грамматики

Table

TestETF

TestBIF

Метками :LL_BIF, :LR_BIF и :NR_BIF помечены переводы описателей соответствующих грамматик в Итерационную Форму Бекуса, получение разборщиков, выполнение тестовых примеров и проверка их в части с меткой :COMPARE.

:BIF_CYCLE - показывает неизменность описателей грамматик при многократном разборе.

:F_BNF, :B_BNF и :I_BNF – преобразование описаний разборщиков грамматик в НФБ.

:I_BIF_LLk – проверка разборщика IGrammarLLk.bif, который представлен нисходящим разбором методом LLk(2). Кроме того, проверяется работа режима `SelfParsing = 0x80000000`, который обеспечивает саморазбор грамматики (*.bif).

Simple

C_99

Разбор программ C и C_99. Особенность - наличие двух лексем определения идентификатора TYPE_NAME и IDENTIFIER. Что требует минимальных действий при синтаксисе. Даже при холостом проходе, без построения синтаксического дерева, необходимо строить таблицы определения идентификаторов и имен типов.

Учебный C - подобный язык имеет конструкции, которые являются ошибочными с точки зрения C, поэтому из тестового файла text.cpp был сформирован файл test_c.cpp, в котором не представлены примеры аппарата типов. Файл text_c.cpp содержит примеры описания типов, поэтому он может быть разобран только с использованием специального сканера.

Части :C и :C_99 содержат проверку грамматик соответствующих языков. Они реализованы как псевдодействия при табличном разборе.

Часть :C_YACC проверяет сгенерированную YACC не модифицированную грамматику.

Задачи

Проект содержит места, где работа выполнена не до конца. Такие места могут быть рассмотрены как задание на практических занятиях.

Лексический анализ

- HEX – числа. Сканер разбирает только целые десятичные и восьмеричные числа и десятичные с фиксированной точкой. Нет шестнадцатеричных, двоичных и с плавающей точкой.
- Строковые данные в разбираемой программе.
- Таблица констант (целые, числа, строки); – реализовано.
- Подключаемая таблица ключевых слов. – реализовано.
- Комментарий. – реализовано.
- /* */ Много строчный комментарий.
- #include – Сканер работает только со строкой в памяти и с одним файлом. Желательно реализовать аппарат подключаемых файлов.
- Макросы. Аппарат макрокоманд без параметров очень просто реализуется с использованием аппарата #include.
- Макросы с параметрами
- #ifdef – условная компиляция без логических условий. – реализовано.
- #if defined() - условная компиляция с логическими условиями && и ||.
- Отдельный парсер. Объединение подключения файлов макрокоманд и условной компиляции в отдельную программу, которая создает промежуточный файл.
- Использование FLEX (генерация шаблона, Priority, Syntax). (C99_Grammar.l). – реализовано.
- Многосимвольные символы.

Новый сканер.

Из всех частей проекта сканер имеет меньше всего вариантов реализации – написанный вручную Parse и Lexer, созданный системой Flex на основе описания грамматики C99.

- Построение системы генерации сканеров на основе описания в виде регулярного выражения
- Построения сканера как отдельного прохода с выдачей массива лексем в виде файла с последующим чтением этого массива фазой синтаксиса.

Работы в проекте

Программы разбора проверялись на ограниченном контрольном примере, поэтому необходимы примеры на различные ситуации, как правильные, так и ошибочные.

- Строки в проекте. В проекте для символьных данных используется char* надо перейти на string.
- Примеры и тесты. Проверки на ошибочные ситуации.
- SQL.
    - Поиск текста описания грамматики в формальной форме
    - Чтение графических изображений с сайта ORACLE.
- Различия C и C99. Примеры для различий.
    - Чистые тексты программ

    - Тестовые примеры из других систем построения трансляторов
- Флора OTL, F++.
- Диалоговые формы.
- Откат ошибок.
- Грамматика NL. Доработать описание грамматики.
- Приоритеты и ассоциации для YACC
- Реализация оператора switch

Стековая машина

Конечным результатом работы программ является список команд некой стековой машины или четверок. Сама машина не реализована, нет полной проработки передачи параметров функция.

- Реализация Компоновщика и Загрузчика.
- Реализация Стековой машины.
- Реализация передачи параметров функций.
- Одно и трех адресные машины.

Программирование действий.

Сейчас действия заранее пишутся на C++ в виде функций с минимальными параметрами. Затем в описателях грамматик задаются действия в виде вызова данных функций с передачей им параметров.

Если будет машина, реализующая действия по построению синтаксического и семантического деревьев, то действия можно будет реализовывать непосредственно в грамматиках.

Грамматики

При построении транслятора требуются преобразования и проверки грамматик. Есть алгоритмы, которые работают только с грамматиками, представленными в определенной форме. Есть алгоритмы преобразования грамматик. Задачи - это реализация алгоритмов:

- Левосторонняя рекурсия – реализовано `delete_LeftRecursion()`
- Empty – правила
- Левая факторизация - `make_LeftFactoring()`
- Форма Хомского
- Алгоритм Кока-Янгера-Касами

Генерации модулей

В системах построения трансляторов выходом являются тексты модулей на языках программирования, которые реализуют лексический и синтаксический анализ. В большинстве случаев действия это программные вставки, не контролируемые системой построения.

- Генерация модулей на C++. Заготовку можно взять у YACC.
- Генерация модулей на JAVA.

Проект на JAVA.

Создание варианта проекта на JAVA. Это может быть простое дублирование частей проекта в новой среде программирования, так и переделка структуры данных и самого проекта. Разные реализации могут взаимодействовать передачей данных через файлы.

- Общая архитектура проекта
- Структура данных
- Подключение сторонних модулей.

Проект ANTLR.

Подключение сканера.

Проверка грамматик ANTLR.

Динамический LALR(k)

Перевод " - A-practical-method-f-Philippe-Charle-[ebooksread.com].pdf", "LALR(k).docx"

LALR(A) Lookahead Sets with Varying-Length Strings

Реализация

Подключение LLVM.

Формирование целевого кода в виде LLVM. С возможностью выполнения. Наибольшая трудность с подключением библиотек и операции ввода/вывода.

Графическое представление грамматики.

Синтаксис SQL представлен в виде графических схем, которые строятся на основе грамматический правил, записанных по правилам итерационной формы Бэкуса.

Литература

«Книга Дракона» (Dragon Book): Ахо А.В., Сети Р., Ульман Д.Д. Компиляторы: принципы, технологии и инструменты, Вильямс, 2003, 768 стр.

http://mexalib.com/view/2169

"Q:\LIBRARY\E-books\Программирование\Искусство программирования\Ахо, Сети, Ульман. Компиляторы. Принципы, технологии, инструменты"

«Книга Дракона-2» (Dragon Book-2): «Компиляторы: принципы, технологии и инструменты», 2-е издание, Альфред В. Ахо, Моника С. Лам, Рави Сети, Джеффри Д. Ульман, 1184 стр., ISBN 978-5-8459-1349-4, «ВИЛЬЯМС», 2011

http://www.proklondike.com/books/thobshee/compiler2.html

"Q:\LIBRARY\E-books\Программирование\Искусство программирования\Ахо, Сети, Ульман. Компиляторы. Принципы, технологии, инструменты_2 издание"

Грис, Д.
Конструирование компиляторов для цифровых вычислительных машин / Д. Грис; Пер. с англ. Е. В. Докшицкой [и др.]; Под ред. Ю. М. Баяковского, В. С. Штаркмана. – М.: Мир, 1975. – 544 с.

Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции

Том 1 Синтаксический анализ. Том 2 Компиляция. М: изд-во "Мир", 1978

Классика программирования: Алгоритмы, языки, автоматы, компиляторы. Практический подход. Секреты мастерства

http://www.bolero.ru//index.php?level=4&pid=39006406

Основы конструирования компиляторов
В. А. Серебряков, М. П. Галочкин
Издательство: Эдиториал УРСС
ISBN 5-8360-0242-8, Тираж: 1000 экз., Формат: 60x84/16

Языки программирования: разработка и реализация. 4-е изд.

Пратт Т., Зелковиц М. Издательство: Питер

ISBN:5-318-00189-0

Основные концепции компиляторов

Р. Хантер

в обл. 252 стр., 2002 год
Издательство: Вильямс Серия:
ISBN: 5-8459-0360-2

http://www.williamspublishing.com/Books/5-8459-0360-2.html