Coco/R – Генератор быстрых претрансляторов для компиляторов промышленного качества.

Реализация для языка Ада.

А.Гавва

Декабрь 2004

Аннотация

Формальное описание компилятора служит двум основным целям:

1. формирофание справочного документа, который описывает синтаксис и семантику языка
2. формирование согласованной нотации из которой может быть сгенерирован эффективный компилятор

Таким образом, системы генерации компиляторов в какой-либо степени акцентируют внимание на эти аспекты. Рассматриваемая система, *Coco/R*, в большей степени концентрирует внимание на второй цели, и позволяет генерировать компиляторы не уступающие в эффективности промышленным компиляторам, которые построены и отоптимизированы "в ручную". *Coco/R* генерирует нисходящие рекурсивные синтаксические анализаторы, которые обладают простым механизмом обработки ошибок и лексическим анализатором (сканером) со специальной схемой буферизации входного потока. Следует заметить, что простота и адекватность системы имеет не меньшее значение чем эффективность системы. Как правило, программисты не заинтересованы в использовании инструмента, который не удобен в работе ввиду наличия сложной криптованной нотации и/или чрезмерного множества специальных опций. Это подразумевает, что инструмент должен облегчать работу программиста не внося ограничения в гибкость использования.

Содержание

1 Об этом документе **3**

2 Введение **4**

3 Язык описания компилятора *Cocol/R* **7**
3.1 Общая структура 7
3.2 Спецификация сканера 8
3.2.1 Символы 8
3.2.2 Лексемы 9
3.2.3 Имена 10
3.2.4 Директивы 11
3.2.5 Комментарии 11
3.2.6 Разное 11
3.3 Спецификация синтаксического анализатора 12
3.3.1 Продукции 12
3.3.2 Выражения 13
3.3.3 Обработка ошибок 13
3.3.4 Слабые символы 15
3.3.5 Требования к LL(1) 17

4 Применение *Coco/R* для генерации компилятора **19**
4.1 Интерфейс сканера 19
4.2 Интерфейс синтаксического анализатора 20
4.3 Проверка грамматики 20

5 Рекомендации для опытных пользователей *Coco/R* **22**
5.1 Применение сканера написанного “вручную” 22
5.2 Специальная подгонка частей компилятора 22
5.3 Доступ к предварительно выбранной лексеме 22
5.4 Семантическое управление синтаксическим анализатором 23

1 Об этом документе

Данный документ является модифицированной версией текстов от Professor Moessenboeck. Модификации сделаны с целью адаптации к реализации *Coco/R* для языка Ада.

При подготовке этого документа был использован перевод (C) Constantin E. Climentieff (“drmad@chat.ru”; “http://www.chat.ru/~drmad”), адаптированный к реализации *Coco/R* для языка Модула-2.

Заинтересованным пользователям также рекомендуется обратиться к книге “Compilers and Compiler Generators” (C) P.D. Terry (“p.terry@ru.ac.za”), которая размещена в Internet: “http://www.scifac.ru.ac.za/compilers/”.

2 Введение

Coco/R является программой, которая получает на входе расширенную грамматику языка, в форме РБНФ (РБНФ - расширенная форма Бэкуса-Наура; английская аббревиатура - *EBNF*), и осуществляет генерацию нисходящего рекурсивного синтаксического анализатора и лексического анализатора (сканера) для этого языка. Программист должен предоставить главный модуль, который вызывает синтаксический анализатор, а также семантические модули, которые вызываются из грамматики (например, модули таблицы символов и кодогенератора).

Входной язык *Coco/R* основан на *атрибутивной грамматике*, которая представляет собой формализм для спецификации семантики контекстно-свободных языков. В оригинальной форме атрибутивная грамматика является статическим описанием, которое указывает взаимозависимости между атрибутами символов без указания порядка обработки взаимозависимостей. Множество генераторов компиляторов придерживаются этой нотации. Однако, для реализации эффективных компиляторов атрибутивную грамматику лучше рассматривать как алгоритмическую нотацию. Таким образом, порядок обработки семантических действий определяется порядком текстуального представления семантических действий в описании грамматики. Такая идеалогия также используется некоторыми генераторами компиляторов (включая *Coco/R*).

В оригинале, *Coco/R* является развитием старой версии этой программы (*Coco*). Основное отличие между *Coco* и *Coco/R* заключается в том, что *Coco/R* генерирует нисходящий рекурсивный синтаксический анализатор, а *Coco* - табличноуправляемый синтаксический анализатор. Кроме того, *Coco/R* интегрирует описание сканера и описание синтаксического анализатора, что позволяет избежать проблемы построения интерфейса между генерируемыми частями. Главным нюансом для *Coco* является то, что все атрибуты определяются глобально, что время от времени создает необходимость сохранения значений атрибутов в стеке. Этот нюанс устранен в *Coco/R*: для продукций атрибуты могут быть определены локально.

Следующий пример позволяет ощутить возможный вид описания компилятора (более подробная спецификация языка описания будет обсуждаться далее). Пример показывает трансляцию описаний переменных. Задача заключается в занесении описанных имен в таблицу символов и вычислении адресов для переменных. Правило трансляции описаний переменных может быть выражено контекстно-свободной грамматикой, которая представлена в форме РБНФ:

```
Variable_Description = Identifier {"," Identifier} ":" Type ";"
```

Простое написание этого правила позволяет получить синтаксический анализатор, который способен осуществлять синтаксическую проверку описаний переменных. Для осуществления семантической обработки описаний переменных необходимо решить проблему способа трансляции описаний переменных. Это требует дополнительного анализа следующих вопросов:

- Чем являются семантические значения “`Variable_Description`”,“`Identifier`” и “`Type`”? Другими словами, что должно быть выполнено при распознавании этих символов и какая информация о контексте должна быть представлена для обеспечения распознавании этих символов? Это приводит к введению *атрибутов* символов. Таким образом, атрибутом для “`Identifier`” будет

являться его имя, а атрибутом для “Type” - некоторая информация о типе. “Variable_Description” не требует какого-либо характерного атрибута. Однако, “Variable_Description” нуждается в атрибуте взятом из контекста, то есть, для размещения переменной необходима информация о следующем свободном адресе в пространстве адресов. Атрибуты можно рассматривать как параметры (ввода или вывода) для синтаксических символов. Они обозначаются следующим образом:

```
Identifier <name>
Type <typ>
Variable_Description <addr>
```

- Какие действия необходимо осуществить для трансляции соответствующей конструкции? Эти действия составляются из инструкций языка программирования общего пользования (для данной реализации - Ада) и заключаются в символы “(.” и “.)”. Семантические действия могут присутствовать в любом месте правой части продукции (иначе правила вывода) и выполняются в указанном месте в процессе синтаксического анализа.

Анализ перечисленных вопросов ведет к получению атрибутивной продукции, которая, например, может иметь следующий вид:

```
Variable_Description <addr : in out Long_Integer>
                          (.  obj_1: Symbol_Table.The_Object;
                              obj_2: Symbol_Table.The_Object;
                              typ  : Symbol_Table.The_Type;
                              n    : Long_Integer;
                              a    : Long_Integer;
                              name : String (1..32);                 .)

 = Identifier <name>      (.  obj_1      := Symbol_Table.Find (name)
                              obj_1.link := null;
                              n          := 1;                       .)
  {"," Identifier <name>  (.  obj_2      := Symbol_Table.Find (name)
                              obj_2.link := obj_1;
                              obj_1      := obj_2;
                              n          := n + 1;                   .)
  } ":"
  Type <typ>              (.  addr := addr + n * typ.size;
                              a    := addr;
                              while  obj /= null  loop
                                 a          := a - typ.size;
                                 obj_1.addr := a;
                                 obj_1      := obj_1.link;
                              end loop;                              .)
  ";".
```

Следует заметить, что хотя форма записи является свободной, расположение синтаксической части слева, а семантисеской – справа обладает хорошей выразительностью и читабельностью, поскольку обеспечивает четкое разделение между синтаксисом и семантикой, и и наглядно показывает какое семантическое действие будет

выполняться при распознавании указанного синтаксического символа. Кроме того, следует обратить внимание на то, что продукции также содержат описания локальных переменных, которые необходимы для семантических действий. Дополнительно, могут быть доступны описания, которые указаны глобально или импортированы из других модулей.

Атрибутивная грамматика может рассматриваться как специализированный язык, который предназначен для написания компиляторов (или подобных им программ). Она является сокращенной нотацией для широко известной технологии нисходящего рекурсивного анализа. Хотя ручная реализация препроцессора (*front-end*) компилятора не так сложна, использование нотации, которая подобна показанной выше, может обладть следующими преимуществами:

- Легкость чтения. Синтаксис и семантика четко разделены. Семантические действия не скрываются среди инструкций синтаксического анализа.
- Работа подпрограмм, таких как: получение следующей лексемы от сканера, обработка альтернатив, необязательных фрагментов и итераций, или обработка ошибок, – не должна быть написана явно, а порождается из грамматики.
- Быстрее и надежнее реализовать компилятор используя высокоуровневую нотацию, а не язык программирования общего назначения. В процессе проектирования языка могут быть легко опробованы несколько различных альтернативных конструкций, а их реализации могут быть сохранены как прототипы.
- В то время как при ручной реализации синтаксического анализатора легко утерять контроль над корректностью грамматики (например, отсутствие кольцевых зависимостей продукций или наличие свойств LL(1)), генератор синтаксического анализатора способен легко выполнить необходимую проверку автоматически.

Генераторы компиляторов позволяют программистам, которые не являются опытными разработчиками компиляторов, проектировать и обрабатывать небольшие языки. Примеры и необходимость использования таких небольших языков достаточно многочисленны (от командных языков, до языков описания структур данных в файлах).

3 Язык описания компилятора *Cocol/R*

Описание компилятора может рассматриваться как модуль, содержащий импорт, декларации и грамматические правила, которые определяют лексическую и синтаксическую структуру языка также, как и правила его трансляции в желаемый язык. Словарь *Cocol/R* использует идентификаторы, строки и числа обычным образом:

```
ident  = letter { letter | digit } .
string = '"' { anyButQuote } '"' | "'" { anyButApostrophe } "'" .
number = digit { digit } .
```

Строчные буквы отличаются от прописных. Строки не должны пересекать границ. Ключевыми словами языка *Cocol/R* являются:

```
ANY CASE CHARACTERS CHR COMMENTS COMPILER CONTEXT END FROM
IGNORE NAMES NESTED PRAGMAS PRODUCTIONS SYNC TO TOKENS WEAK
```

Примечание: NAMES - это расширение оригинального синтаксиса языка Оберон.

Следующие метасимволы использованые для формирования выражений РБНФ:

```
(    )       для группировки фрагментов
{    }       для итераций фрагментов
[    ]       для необязательных фрагментов
<    >       для атрибутов
(.   .)      для семантических фрагментов
= . | + -    как описано ниже
```

Комментарии заключаются в "(*" и "*)" и могут быть вложенными. Кроме того, поддерживаются комментарии в стиле языка Ада (не вложенные). Семантические части могут содержать описания или инструкции на языках высокого уровня (в данной реализации - Ада).

3.1 Общая структура

Описание компилятора состоит из следующих частей:

```
Cocol =   "COMPILER" goalIdent
          ArbitraryText
          ScannerSpecification
          ParserSpecification
          "END" goalIdent "." .
```

Имя после ключевого слова "COMPILER" - это имя грамматики и оно должно соответствовать имени после ключевого слова "END". Имя грамматики также означает самый верхний нетерминал (стартовый символ).

После имени грамматики может следовать текст на языке Ада, но это не отслеживается *Coco/R*. Этот раздел обычно содержит спецификаторы контекста и описания

глобальных объектов (констант, типов, переменных и подпрограмм), которые потом потребуются в семантических действиях.

Оставшиеся части описания компилятора специфицируют обрабатываемые лексическую и синтаксическую структуры языка. Целесообразно специфицировать две грамматики - одну для лексического анализатора (сканера) другую для синтаксического анализатора. Нетерминалы (обобщенные лексемы), распознаваемые сканером, рассматриваются разборщиком как терминалы.

3.2 Спецификация сканера

Сканер должен читать исходный текст, пропускать незначащие символы и распознавать лексемы, которые должны быть переданы синтаксическому анализатору. Лексемы могут быть классифицированы как литералы (ключевые последовательности) или классы лексем. Литералы (например, "`END`", "`=`" и пр.) записываются как строки и означают сами себя. Они располагаются в правой части продукций, и их не надо описывать. Обобщенные лексемы (например, идентификаторы или числа) имеют определенную структуру, которая должна быть описана в виде регулярного выражения РБНФ. Существует много различных видов обобщенных лексем (например, множество идентификаторов), которые распознаются как одна и та же лексема.

```
ScannerSpecification = {  "CHARACTERS" { SetDecl }
                        | "TOKENS"     { TokenDecl }
                        | "NAMES"      { NameDecl }
                        | "PRAGMAS"    { PragmaDecl }
                        | CommentDecl
                        | VariousDecl
                        } .
```

Спецификация сканера состоит из 6 необязательных частей, которые могут быть расположены в произвольном порядке.

3.2.1 Символы

Эта секция обеспечивает описания имен для множеств символов, таких как буквы или цифры. Эти имена могут быть использованы в других секциях спецификации сканера.

```
SetDecl   = ident "=" Set ".".
Set       = BasicSet { ( "+" | "-" ) BasicSet }.
BasicSet  = ident
            | string
            | "CHR" "(" number ")" [ ".." "CHR" "(" number ")" ]
            | "ANY".
```

SetDecl присваивает множеству символов имя. Базовое множество символов определяется как:

- "`string`" - множество, состоящее из всех символов строки;

- “ident” - заранее определенное множество символов с его именем;
- “ANY” - множество всех символов;
- “CHR(i)” - отдельный символ с порядковым номером (ординалом) i;
- “CHR(i)..CHR(j)” - множество символов с ординалами между i и j.

Возможность специфицировать диапазон от “CHR(7)” до “CHR(31)” это расширение над оригинальной реализацией языка Оберон.

Множества символов могут быть сформированы из базовых множеств при помощи операторов:

- “+” сложение (объединение) множеств;
- “-” вычитание множеств.

Примеры:

```
digit="0123456789"            Множество всех цифр;
hexdigit=digit+"ABCDEF"       Множество всех 16-ричных цифр;
eol=CHR(13)                   Символ конца строки;
noDigit=ANY - digit           "Не-цифра";
ctrlChars=CHR(1)..CHR(31)     Управляющие символы ASCII.
```

3.2.2 Лексемы

Лексема рассматривается синтаксическим анализатором как терминальный символ, для сканера - это синтаксически структурированный символ. Эта структура должна быть описана регулярным выражением в РБНФ.

```
TokenDecl    = Symbol [ "=" TokenExpr "." ] .
TokenExpr    = TokenTerm { "|" TokenTerm } .
TokenTerm    = TokenFactor { TokenFactor }
               [ "CONTEXT" "(" TokenExpr ")" ] .
TokenFactor  = Symbol
               | "(" TokenExpr ")"
               | "[" TokenExpr "]"
               | "{" TokenExpr "}" .
Symbol       = ident | string .
```

Лексемы могут быть описаны в произвольном порядке. Описание лексемы определяет символ совместно с его структурой. Обычно символ в левой части описания представляет собой идентификатор. Он указывается для того, чтобы раскрыть смысл правой части описания.

Правая часть описания лексемы специфицирует структуру лексемы при помощи регулярного выражения РБНФ. Это выражение может содержать литералы, означающие сами себя (например “END”), так же как и имена множеств символов (например, букв), описывающие произвольный символ из таких множеств. Фраза “CONTEXT” в

TokenTerm означает тот единственный терм, который был распознан, когда его правый контекст во входном потоке - это TokenExpr, заключенный в скобки.

В отдельных случаях левый символ может быть строкой. В этом случае справа может быть ничего не указано. Это используется, когда пользователь желает заниматься разбором "вручную". При этом правая часть описания опускается и сканер не генерируется.

Примеры:

```
ident   =   letter { letter | digit } .
real    =   digit { digit } "." { digit }
             [ "E" [ "+" | "-" ] digit { digit } ] .
number  =   digit { digit }
          | digit { digit } CONTEXT ( ".." ) .
and     =   "&" | "AND" .
```

Фраза "`CONTEXT`" в предыдущем примере обеспечивает различие между вещественными числами (например, 1.23) и диапазонными конструкциями (например, 1..2), что не может быть однозначно оттранслировано, если просматривать вперед только на один символ.

ЗАМЕЧАНИЕ: сканер экспортирует две переменных: "`pos`" (позиция) и "`len`" (длина), - которые представляют собой позицию и длину самой ранней распознанной лексемы. Он также экспортирует процедуру

```
Get_Name
  (pos        : in     INT32;
   len        : in     CARDINAL;
   sourceText :    out FileIO.String_Ptr);
```

которая может быть использована для доступа к действительному содержимому лексемы в позиции "`pos`" длиной "`len`".

3.2.3 Имена

Обычно, сгенерированные сканер и синтаксический анализатор используют беззнаковые целые литералы для обозначения символов и лексем. Это делает синтаксические анализаторы "нечитабельными". Используя директиву компилятора "`$N`" или параметр командной строки "`-n`", можно заставить *Coco/R* генерировать символьные обозначения. По умолчанию, эти имена имеют стереотипную форму. "`NameDecl`" можно использовать для предпочтительных определяемых пользователем имен:

```
NameDecl = Ident  "=" ( ident | string ) "." .
```

Примеры:

```
lss = "<" .
ellipsis = ".." .
```

Возможность использовать имена - это расширение над оригинальной реализацией для языка Oberon.

3.2.4 Директивы

Директива (*pragma*) - это лексема, которая может встретится где-то во входном потоке (это могут быть символы конца строки или опции компилятора). Довольно затруднительно обрабатывать все многочисленные места, в которых директива может встретиться в грамматике. Поэтому существует специальный механизм для обработки директив без включения их внутрь продукций. Директивы описываются подобно лексемам, но им может быть назначено семантическое действие, которое будет выполняться вне зависимости от того, распознаны ли они сканером или нет.

```
PragmaDecl   =  TokenDecl [ SemAction ] .
SemAction    =  "(." arbitraryText ".)" .
```

Пример:

```
option = "$" { letter } .
    (. Scanner.GetName(Scanner.pos, Scanner.len, str);
       i := 1;
       WHILE i < Scanner.len DO
         CASE str[i] OF
         ...
         END;
         INC(i)
       END .)
```

3.2.5 Комментарии

Комментарии трудно (а вложенные - даже невозможно) описать в терминах регулярных выражений. Это вызывает необходимость иметь специальнную конструкцию для обработки подобной структуры. Комментарии описываются специфицированием их открывающих и закрывающих ограничителей. Возможно описывать различные типы комментариев. Ограничители не должны быть длинней двух символов.

```
CommentDecl =
"COMMENTS" "FROM" TokenExpr "TO" TokenExpr [ "NESTED" ] .
```

Примеры:

```
COMMENTS FROM "(*" TO "*)" NESTED
COMMENTS FROM "--" TO lf
```

3.2.6 Разное

Следующие опции помогают гибко генерировать сканер.

```
VariousDecl = "IGNORE" ("CASE" | set).
```

"`IGNORE CASE`" означает, что прописные буквы в именах не отличаются от строчных.

"`IGNORE набор`" определяет набор незначимых символов, которые пропускаются сканером, если он встречает их среди лексем (например, символы табуляции или конца строки). Пробел пропускается по умолчанию.

3.3 Спецификация синтаксического анализатора

Спецификация синтаксического анализатора является главной частью описания компилятора. Она содержит продукции атрибутивной грамматики, специфицирующей синтаксис языка, который необходимо распознать, прежде чем оттранслировать. Продукции могут располагаться в произвольном порядке. Допускаются ссылки на еще неописанные нетерминалы. Любое имя, котрое предварительно не было описано как терминальная лексема, должно рассматриваться как нетерминал.

Для каждого нетерминала должна быть в точности одна продукция (или список альтернативных правых частей). Должна присутствовать продукция для начального символа, соответствующая имени грамматики.

```
ParserSpecification = "PRODUCTIONS" { Production } .
Production          = ident [ FormalAttributes ]
                      [ LocalDecl ] "=" Expression "." .
FormalAttributes    = "<"  arbitraryText ">" .
LocalDecl           = "(." arbitraryText ".)" .
```

3.3.1 Продукции

Продукция может быть рассмотрена как процедура, которая разбирает нетерминал. Она имеет собственную область действия для атрибутов и локальных объектов, и состоит из левой и парвой частей, которые разделены символом равенства. Левая часть специфицирует имя нетерминала совместно с его формальными атрибутами и локальными описаниями. Правая часть состоит из контекстно-свободного выражения РБНФ, которое специфицирует структуру нетерминала, т.е. способ его трансляции. Формальные атрибуты записаны как формальные параметры в языке Ада. Они заключены в угловые скобки. По аналогии со входными и выходными параметрами подпрограмм мы будем использовать термины "*входные атрибуты*" и "*выходные атрибуты*". Локальные описания - это произвольные описания языка Ада, заключенные в "`(.`" и "`.)`". Продукция устанавливает область действия для своих формальных атрибутов и их локально описанных объектов. Терминалы и нетерминалы, в общем случае называемые объектами, а также импортированные программные модули видны в любой продукции.

Пример:

```
Expression <x: in out Item>          (* параметры *)
           (. y       : Item;
              operator: Integer; .)  (* локальные переменные *)
= (* описание выражения *) .
```

3.3.2 Выражения

Выражение РБНФ описывает контекстно-свободную структуру некоторой части исходного языка совместно с атрибутами и семантическими действиями, которые специфицируют трансляцию этой части в требуемый язык.

```
Expression   = Term { "|" Term } .
Term         = { Factor } .
Factor       = [ "WEAK" ] Symbol [ Attributes ]
               |  SemAction
               |  "ANY"
               |  "SYNC"
               |  "(" Expression ")"
               |  "[" Expression "]"
               |  "{" Expression "}" .
Attributes   = "<"  arbitraryText ">" .
SemAction    = "(." arbitraryText ".)" .
Symbol       = ident | string .
```

Нетерминалы могут иметь атрибуты. Они записываются как действительные параметры языка Ада и заключаются в угловые скобки. Если нетерминал имеет формальные атрибуты, каждое появление этого нетерминала должно иметь список действительных атрибутов, который соответствует формальным атрибутам в соответствии с правилами совместимости языка Ада. Соответствие, тем не менее, проверяется только тогда, когда осуществляется компиляция модуля синтаксического анализатора. Семантическое действие - это произвольная последовательность инструкций языка Ада, заключенная в "`(.`" и "`.)`". Символ "`ANY`" обозначает любой терминал, который не является альтернативой символу "`ANY`". Обычно он используется для разбора структур, которые содержат произвольный текст. Например, трансляция списка атрибутов *Cocol/R* в сущности такова:

```
Attributes < pos, len: in out Long_Integer > =
    "<"                (. pos := Scanner.pos + 1 .)
    { ANY }
    ">"                (. len := Scanner.pos - pos .) .
```

В этом примере закрывающая угловая скобка - это неявная альтернатива сивола "`ANY`" в фигурных скобках. Смысл в том, что "`ANY`" соответствует любому терминалу, за исключением "`>`". "`Scanner.pos`" - это позиция в исходном тексте для последней распознанной лексемы. Она экспортируется сгенерированным сканером.

3.3.3 Обработка ошибок

Правильное и эффективное восстановление после ошибок трудноосуществимо в синтаксических анализаторах, которые основаны на идее нисходящего рекурсивного синтаксического анализа, поскольку при возникновении ошибки, доступно не достаточно информации о состоянии процесса синтаксического анализа. Что необходимо сделать в случае ошибки:

1. найти все символы, с которыми синтаксический анализ может быть продолжен с определенной позиции в доступной грамматике, начиная с позиции ошибки (восстановимые символы).

2. пропустить весь входной поток до появления первого символа, входящего во множество восстановимых.

3. переместить синтаксический анализатор в позицию, где восстановимый символ может быть распознан.

4. продолжить синтаксический анализ с этой позиции.

В синтаксических анализаторах, которые основаны на нисходящем рекурсивном синтаксическом анализе, информация о позиции разбора и об ожидаемых символах неявно содержится в коде синтаксического анализатора (и в процедуре call stack), и не может быть использована для восстановления после ошибок. Один из методов преодолеть это заключается в динамическом вычислении восстановимого множества во время разбора. В дальнейшем, если возникает ошибка, восстановимые символы уже известны и все, что необходимо сделать, это пропустить ошибочный входной поток и "раскрутить" процедуру "`stack up`" до допустимой точки продолжения ([Wirth76]). Этот подход, хоть и автоматизируется, но замедляет безошибочеый разбор и увеличивает объем кода синтаксического анализатора.

Другой подход был упомянут в [Wirth86]. В восстановлении принимают участие только некоторые синхронизирующие точки грамматики. Ошибки в других точках обнаруживаются, но не подлежат восстановлению. Разбор просто продолжается со следующей синхронизирующей точки, где грамматика и входной поток снова соответствуют друг другу. Это требует, чтобы разработчик грамматики специфицировал точки синхронизации единственным образом - не слишком сложная работа, если немножко подумать. Преимущество этой идеи заключается в том, что никаких множеств восстанавливаемых символов не надо вычислять в процессе работы. Это делает разборщик маленьким и быстрым.

Программисту необходимо учитывать несколько рекомендаций для того, чтобы *Coco/R* генерировал хороший и эффективный обработчик ошибок.

Во-первых, необходимо специфицировать точки синхронизации. Точки синхронизации - это позиции в грамматике, где ожидаются особо надежные терминалы, которые трудно пропустить или неправильно оформить. Когда генерируемый синтаксический анализатор достигает такой точки, он подгоняет входной поток к следующему ожидаемому в этой точке символу. Во многих языках хорошими кандидатами на точки синхронизации являются начала инструкций (где ожидается что-то вроде IF, WHILE и т.п.), начала секций описания (где ожидается что-то вроде CONST, VAR и пр.). Символ конца файла всегда один из символов синхронизации, гарантирующий, что синхронизация прекращается как минимум в конце исходного текста. Точки синхронизации специфицируются символом "`SYNC`".

Точка синхронизации транслируется в цикл, который пропускает все символы, не ожидаемые в этой точке (исключая конец файла). Множество таких символов может быть предварительно вычислено на этапе генерации синтаксического анализатора. Следующий пример показывает две точки синхронизации и их двойников в сгенерированном синтаксическом анализаторе:

```
PRODUCTION                    SPIRIT OF GENERATED PARSING PROCEDURE
Declarations =
   SYNC                       while  not (sym Is_In (...)  loop
                                                  -- const, type, var, proc,
                                                  -- begin, end, eof
                                 Error (...);
                                 Get;
                              end loop;
{                             while  sym Is_In (...)  loop
                                             -- const, type, var, proc
  ( "CONST" { ConstDecl ";" }   if  sym = const  then
                                   Get;...
  | "TYPE" { TypeDecl ";" }     elsif  sym = type  then
                                   Get;...
  | "VAR" { VarDecl ";" }       elsif  sym = var  then
                                   Get;...
  | ProcDecl                    else
                                   ProcDecl
                                end if;
  )                           end loop;
  SYNC                        while  not (sym Is_In (...)  loop
                                                  -- const, type, var, proc,
                                                  -- begin, end, eof
                                 Error(...);
                                 Get;
                              end loop;
}
```

Чтобы избежать ложных сообщений об ошибках, о наличии ошибок сообщается лишь когда определенная часть текста после последней ошибки будет корректно проанализирована.

3.3.4 Слабые символы

Обработка ошибок может быть выполнена посредством спецификации этих терминалов как “слабых” в текущем контексте. Слабый теминал - это символ, в котором можно допустить ошибку или пропустить его, как в точке с запятой между инструкциями. Слабый терминал отмечается записью перед ним ключевого слова “WEAK”. Когда сгенернированный синтаксический анализатор не находит слабых терминалов, он прогоняет входной поток до следующего символа, который либо является допустимым наследником слабого символа, либо символом, ожидаемым в любой точке синхронизации (считается, что символы в точках синхронизации настолько “сильные”, что их никогда невозможно пропустить).

ПРИМЕР:

```
StatementSeq = Statement { WEAK ";" Statement } .
```

Если синтаксический анализатор пытается распознать слабый символ и обнаруживает, что он пропущен, он информирует об ошибке и пропускает входной поток до

тех пор, пока правильный наследник не будет найден (или символ, который ожидается в какой-нибудь точке синхронизации; избегать пропуск надежных символов можно лишь эвристически). Следующий пример демонстрирует трансляцию слабого символа.

Идея генерируемого кода синтаксического анализатора:

```
Statement =
   ident                Expect (ident);
   WEAK ":="            Expect_Weak (becomes,
                                     {стартовый символ для Expression});
   Expression           Expression
```

Процедура "Expect_Weak" в общих чертах реализована следующим образом:

```
procedure Expect_Weak
  (s       : Integer;
   expected: Set)
is
begin
   if  sym = s  then
      Get
   else
      Error (s);
      while  not sym Is_In (expected +
                            {символы, ожидаемые в
                             точках синхронизации})  loop
         Get
      end loop;
   end if;
end Expect_Weak;
```

Слабые символы дают синтаксическому анализатору еще один шанс для синхронизации в случае ошибки. Кроме того, множество ожидаемыхх символов может быть заранее вычислено на этапе генерации синтаксического анализатора и не вызывать замедления его работы при безошибочном разборе. Когда итерация начинается со слабого символа, этот символ называется слабым разделителем и обрабатывается особым образом. Если он не может быть распознан, входной поток пропускается до тех пор, пока не будет обнаружен символ, содержащийся в одном из трех множеств:

- символы, которые могут следовать за слабым разделителем
- символы, которые могут следовать за итерацией
- символы, ожидаемые в любой точке синхронизации (включая eof)

Следующий пример демонстрирует трансляцию слабого разделителя:

ГЕНЕРИРУЕМАЯ ПРОЦЕДУРА РАЗБОРА

```
StatSequence =
```

```
Stat               Stat;
{ WEAK ";" Stat }. while  Weak_Separator (semicolon, A, B)  loop
                      Stat;
                   end loop;
```

В этом примере, “A” - это можество стартовых символов инструкции (идентификатор, IF, WHILE и пр.), а “B” - это множество наследников последовательности инструкций (END, ELSE, UNTIL и пр.). Оба множества вычисляются предварительно, на этапе генерации синтаксического анализатора. “Weak_Separator” реализуется следующим образом:

```
function Weak_Separator
  (s: Integer;
   sySucc: Set
   iterSucc: Set)
  return Boolean
is
begin
   if  sym = s  then
      Get;
      return TRUE;
   elsif  sym Is_In (iterSucc)  then
      return FALSE;
   else
      Error(s);
      while  not sym Is_In (sySucc + iterSucc + eof)  loop
         Get;
      end loop;
      return sym Is_In (sySucc) -- TRUE подразумевает 'вставку s'
   end if;
end Weak_Separator;
```

Можно обратить внимание на то, что множество “B” содержит наследников для последовательности инструкций в любом допустимом контексте. Это множество может быть очень большим. Если последовательность операторов встречается в инструкции цикла (“loop”), то допустимый наследник - только “end loop”, но не “end” или “else”. Мы согласны с этим замечанием до тех пор, пока возможно предварительно вычислить множество “B” на этапе генерации синтаксического анализатора. Появление “end” или “else” очень маловероятно в этом контексте, и может быть причиной некорректной синхронизации, что потребует от синтаксического анализатора пересинхронизироваться заново.

3.3.5 Требования к LL(1)

Нисходящий рекурсивный синтаксический анализ требует, чтобы грамматика языка удовлетворяла свойства LL(1). Это означает, что в любой точке грамматики синтаксический анализатор должен быть способен принять решение на основе единственного впереди идущего символа из избранного множества возможных альтернатив. Например, следующая продукция не относится к LL(1):

```
Statement = ident ":=" Expression
          | ident [ "(" ExpressionList ")" ] .
```

Здесь, обе альтернативы начинаются с символа “`ident`”, и синтаксический анализатор не может сделать выбор между продолжением идти по инструкции и поиском “`ident`” в качестве следующего входного символа. Тем не менее, эта продукция может быть легко трансформирована в:

```
Statement = ident (   ":=" Expression
                  |  [ "(" ExpressionList ")"  ]
                ) .
```

где все альтернативы начинаются с различных символов. Существуют конфликты LL(1), которые не так легко распознать, как в предыдущем примере. Для программиста может быть тяжело найти их, если у него нет инструмента для проверки грамматики. В результате должен получиться синтаксический анализатор, который, в некоторых ситуациях, выбирает неверную альтернативу. *Coco/R* проверяет грамматику на соответствие свойствам LL(1), и выдает соответствующие сообщения об ошибках, что помогает скорректировать нарушение.

4 Применение *Coco/R* для генерации компилятора

Атрибутивная грамматика является основополагающим документом для построения реализации компилятора с помощью *Coco/R*. При этом пользователь должен решить следующие задачи:

1. написать атрибутивную грамматику
2. при необходимости, написать семантические модули (пакеты языка Ада) и импортировать их в атрибутивную грамматику (т.е. указать соответствующие спецификаторы "`with`" и "`use`")
3. запустить на выполнение *Coco/R* для генерации сканера, синтаксического анализатора и главной процедуры компилятора, согласно указанной атрибутивной грамматики
4. при необходимости, написать модуль, который будет осуществлять вызов главной процедуры компилятора

Команда

```
acr [-options] grammar_name [-options]
```

осуществит трансляцию описания компилятора расположенного в файле "`grammar_name.atg`" (с именем грамматики, например, *G*) в пакет сканера (файлы: "*g*`s.ads`" и "*g*`s.adb`"), пакет синтаксического анализатора (файлы: "*g*`p.ads`" и "*g*`p.adb`") и главную процедуру компилятора (файл: "*g*`.adb`").

Чтобы получить информацию об используемых опциях запуска *Coco/R* можно выполнить команду

```
acr -?
```

4.1 Интерфейс сканера

Интерфейс сканера описывается в файле спецификации "*g*`s.ads`". Процедура "`Reset`" вызывается из синтаксического анализатора "`Source`" для инициализации сканера. Следует заметить, что ответственность за открытие файла входного потока возложена на главную процедуру компилятора (или модуль пользователя, который осуществляет ее вызов). После вызова "`Reset`", синтаксический анализатор повторно вызывает процедуру "`Get`", для получения из входного потока последующих лексем. Информация о последней распознанной лексеме может быть получена из переменных "`pos`", "`line`", "`col`" и "`len`". Процедура

```
procedure Get_Name
  (The_Position   : in     INT32;
   The_Length     : in     CARDINAL;
   The_Name       :    out FileIO.String_Ptr);
```

может быть использована для получения текста лексемы в позиции "`The_Position`" длина которой "`The_Length`".

При каждой синтаксической ошибке синтаксический анализатор вызывает процедуру ссылка на которую расположена в переменной "`Error`". Пользователь может установить ссылку на процедуру, которая сразу выдает сообщение об ошибке или накапливает информацию об ошибках для более позднего вывода (при этом, профиль процедуры пользователя должен соответствовать типу "`Error_Procedure`"). Процедура "`Error`" также может быть использована для выдачи сообщений о семантических ошибках. При этом следует убедиться, что номера семантических ошибок не конфликтуют с номерами синтаксических ошибок. Для этого, например, следует назначать номера семантических ошибок начиная с 200. Номера ошибок с сопутствующим текстом сообщения помещаются в код генерируемого синтаксического анализатора, и могут иметь следующий вид:

```
when   0 => Msg ("EOF expected");
when   1 => Msg ("identifier expected");
when   2 => Msg ("string expected");
. . .
```

Следует заметить, что для генерации сканера *Coco/R* использует файл шаблона "`scanner.frm`", который, при необходимости, может быть отредактирован пользователем. Файл шаблона должен располагаться в текущем каталоге или в каталоге, который указывается переменной окружения "`CRFRAMES`".

4.2 Интерфейс синтаксического анализатора

Интерфейс синтаксического анализатора описывается в файле спецификации "*g*`p.ads`". Основной подпрограммой синтаксического анализатора является процедура "`Parse`". Главная подпрограмма генерируемого компилятора должна предварительно открыть файл входного потока и вызвать процедуру "`Parse`".

Для генерации синтаксического анализатора и тела главной процедуры генерируемого компилятора *Coco/R* использует файлы шаблона "`parser.frm`" и "`compiler.frm`" соответственно. Эти файлы шаблонов должны располагаться в текущем каталоге или в каталоге, который указывается переменной окружения "`CRFRAMES`".

4.3 Проверка грамматики

Coco/R автоматически проверяет корректность грамматики и информирует пользователя о наличии в ней ошибок.

Генерация компилятора не производится в случаях когда:

- обнаружено использование нетерминала для которого не указана соответствующая продукция
- присутствует продукция для нетерминала, который недостижим из стартового символа
- при обнаружении нетерминала, который не может быть представлен как последовательность нетерминалов
- обнаружены кольцевые взаимозависимости для нетерминалов

- при наличии описаний терминальных символов с одинаковой структурой

Кроме того, *Coco/R* осуществляет проверку и информирует пользователя об обнаружении в грамматике конфликтов LL(1). Однако, наличие конфликтов LL(1) не всегда приводит к генерации неработоспособного компилятора, а в некоторых случаях это может служить индикацией какого-либо желаемого эффекта. В таких случаях необходимо учитывать, что при наличии конфликта LL(1) для какой-либо конструкции "`X`", генерируемый синтаксический анализатор выберет первую альтернативу, которая возможна для конструкции "`X`".

5 Рекомендации для опытных пользователей *Coco/R*

5.1 Применение сканера написанного “вручную”

Задача лексического анализа, которая возложена на сканер, достаточно затратна по времени. Хотя сканер, который генерирует *Coco/R*, является оптимизированным, его реализация использует детерминированный конечный автомат, что влечет за собой дополнительный расход производительности. Таким образом, при повышенных требованиях к времени выполнения приложения, может быть желательно использование синтаксического анализатора, который сгенерирован с помощью *Coco/R*, совместно со сканером, который реализован “вручную”. Это можно осуществить, путем описания в файле атрибутивной грамматики всех терминальных символов (включая используемые литералы) как лексем, но при этом не указывая структуру терминальных символов с помощью РБНФ. Например:

```
TOKENS
  ident
  number
  "IF"
  . . .
```

Когда именованные лексемы описаны без указания структуры, сканер не генерируется. Лексемам назначаются номера в порядке их описания (то есть, первая лексема получает номер 1, вторая - 2, и т.д.). Номер 0 резервируется для символа окончания файла. Сканер написанный “вручную” должен возвращать номера лексем в соответствии с этим соглашением, и должен обладать рассмотренным ранее интерфейсом (для более полного примера можно обратиться к исходным текстам *Coco/R*, и посмотреть содержимое файла “`crs.ads`”, а также посмотреть содержимое файла шаблона для сканера “`scanner.frm`”).

5.2 Специальная подгонка частей компилятора

Использование инструмента для генерации компилятора, как правило, повышает производительность программиста. Однако, это в некоторой степени накладывает ограничения на генерируемый компилятор и гибкость его применения. Исходя из этого, всегда существуют специфические случаи, которые могут быть обработаны более эффективно при использовании реализации компилятора, которая написана “вручную”. Хороший инструмент, кроме предопределенного способа решения задач, в случае необходимости, способен предоставить пользователю возможность решения специфических задач. Как уже ранее упоминалось, *Coco/R* осуществляет генерацию сканера, синтаксического анализатора и главной процедуры компилятора на основе текстовых шаблонных файлов “`scanner.frm`”, “`parser.frm`” и “`compiler.frm`”. При необходимости, программист может отредактировать содержимое этих файлов и изменить внутренние алгоритмы, в зависимости от нужд решаемой задачи (например, для изменения схемы буферизации входного потока).

5.3 Доступ к предварительно выбранной лексеме

Ранее, при обсуждении интерфейса генерируемого сканера, организация доступа к предварительно выбранной лексеме не рассматривалась. Однако, интерфейс гене-

рируемого сканера предусматривает переменные, с помощью которых обеспечивается доступ к предварительно выбранной лексеме:

```
nextPos  :  INT32;      -- file position of lookahead symbol
nextLine :  Integer;    -- line of lookahead symbol
nextCol  :  Integer;    -- column of lookahead symbol
nextLen  :  CARDINAL;   -- length of lookahead symbol
```

Перечисленные выше переменные указывают на последнюю *отсканированную* лексему, а переменные:

```
pos      :  INT32;      -- file position of current symbol
line     :  Integer;    -- line and column of current symbol
col      :  Integer;
len      :  CARDINAL;   -- length of current symbol
```

указывают на последнюю *синтаксически проанализированную* лексему.

5.4 Семантическое управление синтаксическим анализатором

В идеале, синтаксический анализ не должен зависеть от семантического анализа (обработки таблицы символов, проверки типов и т.д.). Однако, некоторые языки программирования обладают конструкциями, которые могут быть обработаны только при наличии семантической информации (например, тип анализируемого символа). Для демонстрации сказанного, рассмотрим пример описания обозначения "`Designator`" для языка Оберон:

```
Designator = Qualident { "." ident        |
                         "^"              |
                         "[" ExprList "]" |
                         "(" Qualident ")"  }.
```

где "`x(T)`" подразумевается как защита типа (т.е. "`x`" объявляется как значение типа "`T`"). Обозначение "`Designator`" может быть использовано в инструкции "`Statement`":

```
Statement = . . .                            |
            Designator [ "(" ExprList ")" ] |
            . . .                              .
```

Здесь, "`x(T)`" можно интерпретировать как обозначение "`x`" (имя процедуры) и параметр "`T`". Две интерпретации "`x(T)`" могут быть распознаны только при анализе типа "`x`". Когда "`x`" является процедурой, открывающая скобка "`(`" является началом списка параметров. В противном случае, открывающая скобка "`(`" является защитой типа.

Coco/R позволяет из семантических действий управлять синтаксическим анализатором. Так, например, обозначение "`Designator`" можно обработать следующим образом:

```
Designator < x : in out Item > =
  Qualident < x >
```

```
{ . . .
|                          (. if  x является процедурой  then
                                 return;
                              end if;                          .)
  "(" Qualident < y > ")"   (. обработать защиту типа          .)
}.
```

Когда откравающая скобка “(” обнаружена после “Qualident”, осуществится выбор альтернативы, которая стартует с открывающейся скобки. Первое семантическое действие этой альтернативы выполнит проверку типа “x”. Если “x” является процедурой, то синтаксический анализатор выполняет возврат из продукции и продолжит обработку в продукции “Statement”.

Список литературы

[Wirth76] Algorithms + Data Structures = Programs, N.Wirth, Prentice-Hill 1976

[Wirth86] Compilerbau. 4th edition., N.Wirth, Teubner Studienbucher 1986