Санкт-Петербургский государственный университет
информационных технологий, механики и оптики

ECG

Генератор компиляторов

Курсовая работа

Е. А. Цымбалюк, студент гр. 5539

2007

Содержание

Введение

Генератор компиляторов *ECG* представляет собой универсальный инструмент для преобразования определенных пользователем структур в программное представление. Определение пользовательских структур включает в себя правила, описывающие саму структуру данных, пользовательские процедуры, вызываемые при выполнении этих правил, а также низкоуровневую процедуру, обеспечивающую базовый ввод данных.

ECG генерирует набор классов на языке *C++*, осуществляющих:

- преобразование входного потока данных в логические переменные «токены» (это выполняет класс лексического анализатора);
- обработку входных данных (это выполняет класс, называемый «парсером»).

В процессе разбора данных парсер вызывает функции лексического анализатора для получения очередного токена, а также вызывает пользовательские процедуры, ассоциированные с определенными пользователем правилами. Совокупность правил называется грамматикой.

ECG реализован на языке *C++*. Программная часть *ECG*, отвечающая за разбор пользовательской грамматики, сгенерирована самим *ECG*.

В основе грамматики лежат описания правил, например:

```
timer -> minutes ":" seconds "." milliseconds;
```

Здесь `timer`, `minutes`, `seconds`, `milliseconds` – «нетерминалы», логические элементы, описываемые правилами грамматики.

Например, следующее значение таймера, соответствует описанному правилу:

```
12 : 59 . 123
```

Правило описания `minutes` может быть, например:

```
minutes -> $number
$number = \d+
```

Здесь `$number` – терминал, описанный регулярным выражением `\d+` (последовательность из одной и более цифр). В процессе разбора входных данных лексический анализатор позволяет определить, что заданная последовательность цифр является числом. Таким образом, парсер на вход получает единственный токен `$number` со значение `12`.

Входные данные могут не соответствовать правилам грамматики. Парсер предоставляет возможность обработать ошибку и восстановить состояние разбора для

продолжения работы. Некорректные данные могут быть исправлены или пропущены – в зависимости от пользовательской реализации. Обработка ошибок осуществляется правилами грамматики.

В некоторых случаях *ECG* не может сгенерировать программу по заданной грамматике. Например, такое может случиться, если грамматика противоречива или неоднозначна. Проблема может быть в неверно выбранном дизайне грамматики, либо описание входных данных не может быть выражено в рамках грамматики *LALR*-класса. В последнем случае часть работы парсера должна быть вынесена на уровень лексического анализатора. *ECG* предоставляет возможность изменять контекст лексического анализатора (может быть изменен набор обрабатываемых токенов). Это позволяет обрабатывать контекстно-зависимые данные незаметно для парсера, расширив при этом возможности грамматики.

Следующие главы описывают основной процесс описания грамматики для *ECG*. В разд. 1 рассмотрены базовые правила описания грамматики. В разд. 2 – способ задания пользовательских семантических процедур, ассоциированных с правилами грамматики. В разд. 3 обсуждается структура генерируемого парсера. В разд. 4 описан лексический анализатор *ECG* и поддерживаемые им регулярные выражения, задаваемые для терминалов. В разд. 5 обсуждаются ошибки, которые могут возникнуть в процессе написания грамматики. В разд. 6 обсуждаются способы разрешения неоднозначностей, возникающих при описании арифметических операторов разного приоритета. В разд. 7 описан механизм смены множества терминалов, обрабатываемых лексическим анализатором, при контекстно-зависимом разборе данных. В разд. 8 – описание механизма типизации терминалов и нетерминалов, обеспечивающего во время компиляции проверку соответствия семантических процедур грамматике. Разд. 9 описывает автоматическое определение типов. В разд. 10 рассмотрен вопрос обработки ошибок и восстановления состояния разбора после них. В разд. 11 описаны сервис вывода сообщений и вспомогательные классы *ECG*.

В приложении A приведен пример грамматики для калькулятора. В приложении B рассмотрен пример калькулятора с поддержкой обработки ошибок. В приложении C приведены грамматики парсера грамматики *ECG* и парсера регулярных выражений.

1. Правила грамматики

Грамматика состоит из набора правил. Правила состоят из терминалов и нетерминалов. Имена, начинающиеся с символа `$` (например, `$number`), либо литералы, записанные в двойных кавычках `“”` (например, `“keyword”`), задают имя терминала. Все остальные имена – имена нетерминалов.

Грамматика *ECG* начинается со следующей секции:

```
#name <имя_класса_парсера> <имя_пользовательского_класса>
```

Первое имя задает имя класса парсера, который будет сгенерирован в ходе обработки грамматики. Второе имя задает имя пользовательского класса, реализующего семантические процедуры.

Пробельные символы при разборе грамматики игнорируются. *ECG* предоставляет строчные `//` и блочные `/*`, `*/` комментарии (как в языке *C++*).

Грамматическое правило имеет вид:

```
N -> RULE ;
```

где `RULE` представляет собой последовательность из нуля или более терминалов и нетерминалов. Символы `->` и `;` – пунктуация *ECG*.

Имена нетерминалов должны состоять из символов английского алфавита, цифр (первой символ имени не может быть цифрой) и символа подчеркивания. При описании терминалов в двойных кавычках используются следующие *escape*-последовательности:

- `\\` – обратный слеш `\`;
- `\n` – перевод строки;
- `\”` – двойные кавычки.

Для того чтобы не переписывать левую часть правила, пользователь может использовать оператор `|`. Это позволяет описать одновременно несколько правил для одного и того же нетерминала:

```
A -> B C ;
A -> D ;
```

Может быть записано в одно правило:

```
A -> B C | D ;
```

Грамматика *ECG* также поддерживает вспомогательные операторы, облегчающие написание и читабельность грамматики:

- ноль или более повторений заданной последовательности:

```
A -> ;
A -> A B C;
```

аналогично:

```
A -> B C * ;
```

- одна и более повторений заданной последовательности:

```
A -> B C;
A -> A B C;
```

аналогично:

```
A -> B C + ;
```

- ноль или одно повторение последовательности:

```
A -> ;
A -> B C ;
```

аналогично:

```
A -> [ B C ] ;
```

- повторяющаяся последовательность, разделенная одним и более элементом:

```
A -> B C ;
A -> A “,” B C ;
```

аналогично:

```
A -> B C % “,” ;
```

Пользователь может использовать круглые и квадратные скобки для выделения группы нетерминалов. При этом будет создан анонимный нетерминал с соответствующим правилом. Созданный нетерминал будет помещен в правую часть исходного правила:

```
A -> (B *) [C % D];
```

преобразуется в:

```
A0 -> B *;
A1 -> C % D;
A1 -> ;
A -> A0 A1;
```

Стартовым нетерминалом является левая часть самого первого правила грамматики. Пользователь может задать стартовый нетерминал явно с помощью директивы `#start`:

```
#start program
```

2. Семантические процедуры

ECG предоставляет возможность задавать семантические процедуры для грамматических правил. Семантические процедуры вызываются в процессе разбора входных данных. С помощью этих процедур происходит «смысловая» обработка текста, благодаря чему отдельные фразы языка приобретают семантику.

В *ECG* семантические процедуры могут возвращать значения и использовать значения, возвращаемые процедурами других правил.

Для описания семантических процедур *ECG* предоставляет два набора парных скобок `{`, `}` и `<<<`, `>>>`, в которых должно быть задано имя процедуры:

```
A -> B "+" C ; <<< add_numbers >>>
```

ECG позволяет передавать процедурам константные параметры. Константы описываются в скобках после имени процедуры:

```
A -> B "+" C ; <<< Calculate(ADD) >>>
A -> B "-" C ; <<< Calculate(SUB) >>>
```

При описании грамматики может быть использованы пустые правила – так называемые «маркеры». Маркеры позволяют обработать входящую информацию еще до того, как правило будет обработано полностью. Для этого пользователь может описать внутри тела правила семантическую процедуру. Это создаст анонимный нетерминал с единственным «эпсилон»-правилом (пустое правило `A -> ;`), при свертке которого вызовется заданная пользователем процедура:

```
A -> B {before_add} "+" C ; <<< add_numbers >>>
```

Функция `before_add` вызовется после разбора нетерминала `B`, но до обработки терминала `"+"`.

Помимо семантически процедур для правила пользователь может определить процедуру для терминала и нетерминала. Эти процедуры будут вызваны при обработке соответствующих элементов в ходе работы парсера. Если элементов для данной процедуры несколько, то элементы могут быть перечислены через символ `|`:

```
$number | "+" | B | C <<< process_element >>>
```

Стоит отметить, что маркеры ослабляют грамматику. Это может привести к возникновению конфликтов разбора из-за того, что компилятор должен принять решение о свертке не в конце правила, а в месте использования правила.

Например, правила

```
A -> A “+” C “-“ D;
A -> A “+” C “*” D;
```

будут нормально разобраны компилятором. Однако при использовании маркеров произойдет конфликт:

```
A -> A “+” {add_sub} C “-“ D;
A -> A “+” {add_mul} C “*” D;
```

3. Устройство парсера

При генерации парсера создаются *.h/.cpp*-файлы с исходными кодами на языке *C++*. В заголовочном файле содержится описание класса, от которого должен быть унаследован пользовательский класс, реализующий семантические процедуры. Также, заголовочный файл содержит список терминалов, которые должны подаваться парсеру на вход в процессе разбора входных данных.

В пользовательском классе необходимо реализовать два метода `NextToken()` и `Token()`, осуществляющие переход к следующему терминалу во входном потоке, и предоставляющие доступ к значению текущего терминала. В данных методах производится работа с файлом (либо любым другим источником данных), анализ входного потока и определение текущего токена. Для сгенерированного парсера предопределены два терминала `TOKEN_EOF` и `TOKEN_ERROR`, сигнализирующие об окончании ввода и возникновении ошибки, соответственно.

Для хранения данных, полученных в ходе разбора входного потока, *ECG* предоставляет стек объектов. Парсер содержит переменную `compilerState`, которая используется для извещения парсера о возникновении ошибки. Ошибки могут быть не только грамматические (в этом случае парсер самостоятельно выставляет флаг ошибки), но и семантические, зависящие от контекста разбора. При восстановлении парсера после ошибки, соответствующий флаг ошибки должен быть снят.

4. Лексический анализатор

Пользователь *ECG* может самостоятельно реализовать функции `NextToken()` и `Token()` для преобразования входного потока в терминалы. Однако пользователь может воспользоваться лексическим анализатором, генерируемым *ECG*. Использование лексического анализатора активируется, если пользователь определил хотя бы одно регулярное выражение для терминала. В этом случае необходимо определить регулярные выражения для всех терминалов, имя которых начинается с символа `$`. Регулярное выражение для терминалов в двойных кавычках можно опустить, тогда в качестве регулярного выражения будет выступать текст, описанный в кавычках.

Для описания регулярного выражения используются символы английского алфавита, множества и *escape*-последовательности. Операторы регулярных выражений лексического анализатора *ECG*: `{}()[]*+-|\`. Регулярное выражение состоит из символов английского алфавита и цифр. Пользователь может определять не только одиночные буквы, но множества символов, например:

```
{abc0-9}
```

данное множество задает три буквы `a`, `b`, `c` и цифры в диапазоне от `0` до `9`.

При задании множества можно использовать отрицание:

```
{^abc0-9}
```

Это множество задает все символы кроме букв `a`, `b`, `c` и цифр в диапазоне от `0` до `9`.

Можно использовать отрицание множеств, а также задавать вложенные множества:

```
{\d^3} – числа от 0 до 9, кроме 3;
{^{\d^3}} – все символы кроме цифр от 0 до 2 и от 4 до 9.
```

Оператор `\` позволяет использовать предопределенные множества символов, а также определять зарезервированные символы (операторы лексического анализатора *ECG*).

- `\\`, `\{`, `\}` и т.д. – зарезервированные символы;
- `\c` – произвольный символ (аналогичное правило «все кроме ничего»: `{^}`);
- `\s` – пробельные символы;
- `\S` – все символы, кроме пробельных;
- `\d` – цифры от `0` до `9`;
- `\D` – все символы, кроме пробельных и цифр от `0` до `9`;
- `\b` – пробел;
- `\t` – табуляция;

- \n – перевод строки;
- \a – буквы английского алфавита (строчные и заглавные);
- \y – множество символов: \"'::,.?!~#%^&|+-*/()[]{}<>=.

Операторы * и + позволяют описать повторяющиеся последовательности символов, например a* (ноль и более букв a) и a+ (одна и более букв a). Данные операторы применяются к последнему определенному символу. Если необходимо применить их к последовательности символов, то используются скобки:

(abc)+ (задает последовательность abcabcabc…)

Для логического оператора «или» пользователь может использовать оператор |:

(ab|cd)* (подходит для последовательности abcdcdab…)

Для определения символа, повторяющегося ноль или один раз, используются квадратные скобки:

a[b]c (задает строки abc и ac).

Для определения регулярного выражения используется оператор равенства:

```
$number = \d+
```

Лексический анализатор *ECG* позволяет определять символы, игнорируемые в процессе разбора (например, пробельные символы), а также строчные и блочные комментарии. Для этого пользователь может определить регулярные выражения для символов skip, cline, cbegin, cend.

```
skip            = \s      // игнорировать пробельные символы
cline           = //      // строчный комментарий
cbegin          = /{*}    // блочный комментарий (начало)
cend            = {*}/    // блочный комментарий (конец)
```

Если пользователь определил регулярные выражения, то *ECG* сгенерирует *.h/.cpp*-файлы, описывающие класс лексического анализатора. Пользовательский класс, реализующий семантические процедуры, должен быть унаследован также и от класса лексического анализатора. Лексический анализатор реализует функции NextToken() и Token(), необходимые парсеру. При этом пользователю достаточно реализовать метод GetChar(), осуществляющий чтение очередного символа из источника данных и передачу этого символа лексическому анализатору.

5. Конфликты и неоднозначности

Грамматические правила могут иметь неоднозначности. Например, для следующего правила возникнет грамматический конфликт:

```
expr -> expr + expr ;
```

Данное выражение имеет неоднозначность, так как входные данные:

```
expr + expr + expr
```

могут быть сгруппированы двумя способами:

```
(expr + expr) + expr
expr + (expr + expr).
```

Первый способ называется левоассоциативным, а второй – правоассоциативным.

Данная неоднозначность называется конфликтом «перенос-свертка»: при встрече символа «+» компилятор не может определить, необходимо ли сначала выполнить правило свертки (для левоассоциативных оператор), либо перенести «+» и свернуть правило вторым способом (для правоассоциативных операторов).

Для разрешения подобного конфликта пользователь может явно задать поведение сгенерированного кода. Для этого *ECG* предоставляет ключевые слова `shift` и `reduce`:

```
expr -> expr + expr ; reduce("+")  // левоассоциативный оператор
expr -> expr + expr ; shift("+")   // правоассоциативный оператор
```

В скобках пользователь может задать набор терминалов, для которых действует данное разрешение конфликта. Если скобки опустить, то данное правило будет действовать на все конфликты для данного правила вне зависимости от терминала.

Помимо конфликта «перенос-свертка», существует конфликт «свертка-свертка». Подобный конфликт возникает в случае, когда одна и та же последовательность терминалов соответствует нескольким правилам одновременно. В такой ситуации пользователь может либо переписать грамматику, избежав подобного конфликта, либо, если исправление возможно, – отметить ключевым словом `reduce`, то правило, которое должно быть свернуто в случае конфликта:

```
A -> (B | C) (D | E);
F -> B D; reduce
```

6. Приоритет операторов

Существуют целый класс неоднозначностей, при которых использование стандартных способов разрешения конфликтов слишком неудобно. Практически для всех арифметических операций можно задать приоритеты, определяющие правила свертки (лево- и правоассоциативность).

Можно выделить два базовых правила для бинарных и унарных операторов:

```
expr -> expr OP expr;
```

и

```
expr -> UNAR expr;
```

Данные правила создают множество неоднозначностей. В рамках *ECG* грамматики эти неоднозначности могут быть разрешены с помощью команд `shift` и `reduce`. Однако разрешение таких конфликтов слишком неудобно, громоздко и провоцирует ошибки со стороны пользователя.

Для решения подобных задач *ECG* предоставляет ключевые слова `left` и `right`, для лево- и правоассоциативных операторов. При этом порядок определения команд задает приоритет операторов – операторы с более высоким приоритетом должны быть расположены в конце:

```
right "=";
left "+" "-";
left "*" "/";
left unar;
```

Данное определение задает правоассоциативный оператор присваивания с наименьшим приоритетом, операторы сложения и вычитания с приоритетом большим, чем у оператора присваивания, но меньшим чем у операторов умножения и деления. Самый высокий приоритет имеет унарный оператор.

Для задания приоритета унарного оператора используется вспомогательное слово `unar`, которое передается в качестве аргумента оператору `prior`, изменяющему приоритет конкретного правила.

После задания приоритета операторов следующие правила уже не будут иметь неоднозначностей:

```
expr -> expr "=" expr
      | expr "+" expr | expr "-" expr
      | expr "*" expr | expr "/" expr;
expr -> "-" expr; prior(unar)
```

При разборе конфликтов «перенос-свертка» и «свертка-свертка» *ECG* выбирает поведение парсера в зависимости от приоритетов операторов.

Парсер, сгенерированный на основе грамматики с подобной техникой решения конфликтов, выполняется намного быстрее, грамматик, в которых данные конфликты разрешаются созданием вложенных нетерминалов для каждого приоритета операторов.

7. Контекст лексического анализатора

В ряде случаев невозможно описать структуру входных данных в рамках *LALR*-грамматики, так как одни и те же символы могут трактоваться по-разному в зависимости от контекста. *ECG* предоставляет пользователю возможность определить контекст лексического анализатора. Под контекстом понимается совокупность всех терминалов, которые могут быть получены при разборе входных данных.

Для описания контекста лексического анализатора *ECG* предоставляет ключевое слово `preset`:

```
preset my_preset {
    $my_token
    +all -comment -"+"
}
```

В теле блока `preset` можно описывать терминалы, задавать им регулярные выражения. Так же можно «включать» и «выключать» из контекста уже заданные терминалы (например, команда `-"+"` выключает терминал `"+"` для контекста `my_preset`). Для удобства можно включать терминалы группами, так команда `+all` включает все терминалы, которые описаны в глобальном пространстве имен (не принадлежат ни одному контексту). Команда `-comment` – отключает встроенные терминалы для описания комментариев (выключает комментарии для данного контекста). Также, для включения/выключения терминалов можно использовать имена других контекстов.

В рамках описываемого контекста можно изменять игнорируемые символы (изменять значение `skip`).

Контекст, выставляемый по умолчанию, имеет имя `default`. Можно определить набор терминалов для контекста по умолчанию.

Если пользователь использует контексты, то для лексического анализатора будет сгенерирована функция смены контекста. Пользователь может вызвать эту функцию в процессе работы парсера. Так, например, смена контекста происходит в компиляторе грамматики *ECG* при разборе регулярного выражения. Это необходимо, потому что множество символов регулярного выражения может пересекаться с ключевыми словами языка грамматики, что не является ошибкой.

8. Типы данных

Одним из неудобств, возникающим при поэтапной реализации грамматики, является необходимость синхронизации семантических процедур с правилами грамматики. Если пользователь осуществляет работу со стеком вручную (самостоятельно помещает данные – результаты работы семантических процедур – в стек), то обнаружение логических ошибок возможно лишь в процессе работы программы (как правило, при ее аварийном завершении). Такими ошибками могут быть, например, изменение типа результата работы семантической процедуры, либо изменение числа элементов правила. В первом случае данные в стеке будут преобразованы к неверному типу, что приведет к порче данных. Во втором случае осуществляется некорректная работа со стеком, что так же приводит к неверному преобразованию типов. Стоит отметить, что данные ошибки могут возникать неявно. Изменение логики семантической процедуры для одного правила может повлиять на работу процедур других правил. При большом размере грамматики отслеживать подобные ошибки становится слишком трудно.

Компиляторов *ECG* предоставляет пользователю возможность задать типы элементов грамматики: терминалов и нетерминалов. В этом случае *ECG* сгенерирует вспомогательные функции-обертки, реализующие работу со стеком. Извлеченные из стека данные передаются в семантическую процедуру в качестве аргументов. Таким образом, осуществляется статическая проверка (в момент компиляции) на соответствие грамматики и семантических процедур.

Типы элементов задаются через двоеточие. Элементы одного и того же типа можно сгруппировать, записав их через запятую:

```
$terminal, "term" : string
nonterm : nonterm_type
```

Если тип задан для терминала, то пользователю необходимо определить соответствующий обработчик, преобразующий строковое значение, полученное лексическим анализатором, в указанный тип.

Данные, полученные в результате работы процедуры, помещаются в стек. При очередной свертке правила элементы, составляющие его правую часть и имеющие типы, передаются в качестве аргументов семантической процедуре. Порядок элементов тот же самый, что и в правиле.

Если нетерминал, в который сворачивается правило, тоже имеет тип, то последним аргументом семантической процедуры будет ссылка на результат. После выполнения процедуры результат помещается в стек.

ECG позволяет опустить семантическую процедуру, если результатом свертки является пользовательский тип. В этом случае будет вызван конструктор результирующего объекта, а созданный объект помещен в стек. При необходимости пользователь может передать конструктору константные параметры. Для этого необходимо воспользоваться ключевым словом `construct:`

```
nonterm -> $terminal;  <<< construct(aConstant) >>>
```

ECG предоставляет несколько встроенных типов: `void` – тип элементов по умолчанию, `string` – строковое значение (стандартная строка `std::string` языка *C++*). Пользователю не обязательно задавать функцию преобразования для терминалов, имеющих строковый тип. В этом случае строковое значение текущего терминала будет помещено в стек автоматически.

Помимо `void` и `string` *ECG* поддерживает массивы пользовательских элементов. Для определения массива необходимо использовать символ «*»:

```
array_item : my_type *
```

Типом нетерминала `array_item` будет массив (стандартный массив `std::vector` языка *C++*) элементов `my_type`.

9. Автоматическое определение типов

Как было сказано выше, элементы имеют тип по умолчанию `void`. Соответственно, пользователю нужно описать типы для каждого типизированного терминала и нетерминала. Работу по выставлению типа можно упростить, так как в ряде случаев тип нетерминала можно выставить автоматически. По умолчанию автоматическое определение типов выключено. Используя директиву `#autotype`, пользователь может включить автоматическое определение типов:

```
#autotype
```

Базовое правило для автоматического определения типа: если все правые части правил для данного нетерминала имеют по одному типизированному элементу одинакового типа, то нетерминалу выставляется тат же тип. При этом правила, правые части которых не имеют ни одного типизированного элемента, игнорируются.

Также *ECG* предоставляет дополнительные правила для следующих конструкций:

```
A -> B *;
A -> B +;
A -> B % C; // где C – имеет тип void
```

Для заданных конструкций тип нетерминала `A` будет – массив элементов. Это позволяет кратко записывать грамматические правила. Все типы элементов определяются автоматически:

```
A -> “function” $string “(“ [$string % “,”] “)”; <<< my_proc >>>
```

Для данного определения семантической процедуре `my_proc` будут переданы два аргумента: строковое значение – имя функции, и массив строк – список аргументов.

10. Обработка ошибок

Генератор компиляторов *ECG* предоставляет ряд средств для обработки ошибок и восстановления после них. Обработка ошибок – достаточно сложная задача, так как большинство решений зависят от контекста выполнения. При возникновении ошибки может оказаться необходимым, например, перестроить дерево разбора, удалить или заменить элементы в таблице, а так же расставить по коду программы проверки, предотвращающие дальнейшую генерацию выходных данных.

Как правило, недопустимо останавливать процесс разбора данных при нахождении ошибки. Чаще всего требуется продолжить обработку данных, чтобы выявить и другие синтаксические ошибки. Для этого в *ECG* зарезервирован терминал `$error`, он может быть использован в грамматическом правиле (например, в ожидаемых местах, в которых легко восстановить состояние разбора):

```
statement -> $name "=" $error;
```

При возникновении ошибки парсер изменяет значение текущего терминала на `$error`. Если в текущем состоянии разбора терминал ошибки является ожидаемым символом, то парсер продолжает разбор с учетом нового терминала. Если пользователь не определил правила обработки ошибки для данного состояния, то вызывается функция `Error`. Таким образом, ошибка обрабатывается либо при свертке правила, содержащего терминал ошибки, либо при вызове стандартного метода обработки ошибок.

Иногда недостаточно просто обработать ошибку. Если ошибка произошла, например, в ходе разбора сложного арифметического выражения, дальнейшая часть выражения может быть пропущена, так как дальнейший разбор приведет лишь к ряду других синтаксических ошибок (не являющимися ошибками как таковыми). Для пропуска подобных символов пользователь может выставить флаг `CLEANUP`, переводящий парсер в состояние пропуска терминалов. В этом состоянии парсер отматывает стек разбора до тех пор, пока не встретит правило, допускающее символ `$error`. После нахождения такого правила парсер пропускает все терминалы, не соответствующие правилу.

Например, для нижеприведенного правила парсер будет пропускать терминалы до тех пор, пока не встретит символ `";"`:

```
statement -> $error ";";
```

При свертке правила, предназначенного для пропуска символов, пользователю необходимо снять флаг `CLEANUP`, тем самым, переведя парсер в нормальное состояние разбора.

Ошибки могут быть не только грамматическими, но и семантическими, то есть могут возникать в процессе работы семантических процедур. В этом случае пользователь самостоятельно обрабатывает ошибку и при необходимости выставляет флаг `CLEANUP`.

Если восстановление ошибки невозможно, необходимо выставить флаг `FATAL_ERROR`, который прерывает дальнейший разбор данных. Этот флаг выставляется автоматически перед вызовом функции `Error`. Пользователь должен сбросить этот флаг вручную, если состояние разбора было восстановлено. Также, этот флаг будет автоматически сброшен самим парсером при выставлении флага `CLEANUP`.

Пользователь может выставлять и сбрасывать флаги, используя переменную парсера `compilerState`. Вспомогательный класс `ecg::HELPER`, реализующий ряд типовых методов для работы с парсером и лексическим анализатором, предоставляет следующие методы, упрощающие использование флагов:

- `StateOK` – сбрасывает флаг `CLEANUP` и `FATAL_ERROR`, переводя парсер в нормальное состояние разбора;
- `StateCleanUp` – выставляет флаг `CLEANUP` для перехода в состояние пропуска терминалов;
- `IsCleanUp` – возвращает значение `true`, если выставлен флаг `CLEANUP`, используется для проверки, что парсер находится в состоянии пропуска терминалов, для отключения вывода сообщений;
- `StateFatal` – выставляет флаг `FATAL_ERROR`.

11. Сервис сообщений и вспомогательные классы

Во время разбора входных данных, при возникновении ошибок необходимо выводить текстовые сообщения, информирующие пользователя о типе и содержании ошибки. При описании грамматики *ECG* пользователь может задать список сообщений об ошибке:

```
msg {
     E01 = "Unexpected token '{token}';
     E02 = "Arithmetic operator expected, but '{token}' found";
     // …
}
```

Каждому сообщению задается уникальный идентификатор, используемый при выводе ошибок. Для получения текста сообщения необходимо использовать функцию `GetMessageById`.

Для формирования финального сообщения, необходимо выполнить подстановку контекстно-зависимых элементов, указанных в фигурных скобках (например, `{token}`). Сделать это позволяет функция `ecg::Compose`, описанная в заголовочном файл *ecg_message.h*.

Благодаря данному сервису упрощается процесс формирования сообщения об ошибке. Также, все сообщения компилятора сгруппированы в одном месте, а не разбросаны по коду.

В главе 10 был рассмотрен класс `ecg::HELPER`, определяющий функции, работающие с флагами. Помимо описанных функций этот класс реализует методы `GetChar` (требуемый для лексического анализатора), стандартный метод `Error`, выводящий сообщение об ошибке и завершающий процесс разбора, а так же метод, инициализирующий компилятор, сохраняющий указатели на входной поток данных и выходной поток вывода сообщений.

При использовании вспомогательного класса `ecg::HELPER` пользователю достаточно реализовать лишь методы семантических процедур.

Источники

1. *Ахо А., Сети Р., Ульман Д.* Компиляторы. Принцип, технологии, инструменты. М.: Вильямс, 2001.
2. *Агамирзян И. Р.* Система технологической поддержки разработки трансляторов «ШАГ». Подсистема построения анализаторов. Л.: Институт теоретической астрономии АН СССР, 1986.
3. *YACC.* http://dinosaur.compilertools.net/yacc/index.html
4. *Новиков Ф.А.* Частное сообщение.

Приложение А. Калькулятор

Данное приложение описывает полную *ECG* спецификацию грамматики для простого калькулятора, позволяющего выполнять арифметические операции, а также сохранять вычисленные значения, используя переменные.

Грамматика калькулятора:

```
#name CALC_IMPL, CALC

// grammar
calc        -> stat *; // program is a list of statements

stat        -> {LINE} expr ";";           <<< Print >>>

// operators priority
right       "=";
left        "+" "-";
left        "*" "/";
left        unar;

expr        -> $name "=" expr;            <<< Assign >>>
expr        -> expr "+" expr;             <<< Calc(ADD) >>>
expr        -> expr "-" expr;             <<< Calc(SUB) >>>
expr        -> expr "*" expr;             <<< Calc(MUL) >>>
expr        -> expr "/" expr;             <<< Calc(DIV) >>>
expr        -> "-" expr; prior(unar)      <<< Negate >>>

expr        -> "(" expr ")";
expr        -> $number;                   <<< Value(NUMBER) >>>
expr        -> $name;                     <<< Value(NAME) >>>

// regular expressions
skip        = \s
cline       = //
cbegin      = /{*}
cend        = {*}/

$name       = {\a_}{\a\d_}*
$number     = \d+

// terminal's and nontreminal's types
$name,
$number     : string
expr        : "int"
```

Пользовательский класс, реализующий семантические процедуры:

```
// файл calc.h
#ifndef CALC_H
#define CALC_H

#pragma once

#include "calc_impl.h"
#include "calc_impl_lex.h"

namespace ecg {
```

```
class CALC : public HELPER<CALC, CALC_IMPL, CALC_IMPL_LEX>
{
public:
	typedef const std::string & STRING;
	typedef P<int> PINT;

	// реализация оператора присваивания
	void Assign(STRING name, PINT value, PINT & res)
	{
		vars[name] = *value;
		res = value;
		std::cout << "=(" << name << ") ";
	}

	// вывод номера текущей строки
	void LINE (void)
	{
		std::cout << Line() << ": ";
	}

	// распечатка значений переменных и результата выражения
	void Print(PINT value)
	{
		std::cout << "-> " << *value << std::endl;
		for (std::map<std::string, int>::const_iterator
			iter = vars.begin(); iter != vars.end(); ++iter
		) {
			std::cout << "\t" << iter->first << "\t= "
				<< iter->second << std::endl;
		}
	}

	enum CALC_TYPE {
		ADD, SUB, MUL, DIV,
	};

	// реализация арифметических операций
	void Calc(CALC_TYPE type, PINT a, PINT b, PINT & res)
	{
		res = new int();
		switch (type) {
			case ADD: *res = *a + *b; std::cout << "+ "; break;
			case SUB: *res = *a - *b; std::cout << "- "; break;
			case MUL: *res = *a * *b; std::cout << "* "; break;
			case DIV: *res = *a / *b; std::cout << "/ "; break;
		}
	}

	// унарный оператор «минус»
	void Negate (PINT a, PINT & res)
	{
		res = new int(-*a);
		std::cout << "neg ";
	}

	enum VALUE_TYPE {
		NUMBER, NAME,
	};

	// получение значения числа или переменной
	void Value(VALUE_TYPE type, STRING val, PINT & res)
	{
		std::cout << val << " ";
```

```
            if (type == NUMBER) {
                  res = new int(atoi(val.c_str()));
                  return ;
            }

            res = new int(vars[val]);
      }

private:
      // массив значений переменных
      std::map<std::string, int> vars;
};

} // namespace ecg

#endif // CALC_H
```

Если подать приведенной реализации калькулятора на вход текст:

```
a = -5;
inf = 8 / 1;
b = a + 5 * a * (a - 3);
c = a * 5 - -3 * b;
a = b /*block comments*/= c;
some1 = 12 * 6 - 3 / 2 ; // line comments
b = (c = some1 + a * 3) / 5;
```

то будет распечатано:

```
2: 5 neg =(a) -> -5
        a       = -5
3: 8 1 / =(inf) -> 8
        a       = -5
        inf     = 8
4: a 5 a * a 3 - * + =(b) -> 195
        a       = -5
        b       = 195
        inf     = 8
5: a 5 * 3 neg b * - =(c) -> 560
        a       = -5
        b       = 195
        c       = 560
        inf     = 8
6: c =(b) =(a) -> 560
        a       = 560
        b       = 560
        c       = 560
        inf     = 8
7: 12 6 * 3 2 / - =(some1) -> 71
        a       = 560
        b       = 560
```

```
        c       = 560
        inf     = 8
        some1   = 71
8: some1 a 3 * + =(c) 5 / =(b) -> 350
        a       = 560
        b       = 350
        c       = 1751
        inf     = 8
        some1   = 71
```

Для каждой строчки входного текста выведено арифметическое выражение, преобразованное в польскую запись. В конце обработки выражения распечатано состояние аргументов программы.

Приложение B. Калькулятор с обработкой ошибок

В данном приложении рассмотрен пример того же калькулятора, что и в приложении A, но с восстановлением ошибок.

Грамматика калькулятора (серым цветом выделен не изменившийся текст):

```
#name CALC_IMPL, CALC

// grammar
calc  -> stat *; // program is a list of statements

stat  -> {LINE} expr ";";                  <<< Print >>>

// operators priority
right "=";
left "+" "-";
left "*" "/";
left unar;

expr  -> $name "=" expr;                     <<< Assign >>>
expr  -> expr "+" expr;              <<< Calc(ADD) >>>
expr  -> expr "-" expr;              <<< Calc(SUB) >>>
expr  -> expr "*" expr;              <<< Calc(MUL) >>>
expr  -> expr "/" expr;              <<< Calc(DIV) >>>
expr  -> "-" expr; prior(unar)       <<< Negate >>>

expr  -> "(" expr ")";
expr  -> $number;                            <<< Value(NUMBER) >>>
expr  -> $name;                              <<< Value(NAME) >>>

// error handling
stat    -> $error ";";                       <<< StateOK >>>
expr  -> $name "=" $error;           <<< Message(L"ER_2") >>>

// regular expressions
skip        = \s
cline       = //
cbegin      = /{*}
cend        = {*}/

$name       = {\a_}{\a\d_}*
$number     = \d+

// terminal's and nontreminal's types
$name,
$number     : string
expr        : "int"

// messages
msg {
      ER_1 = "Unexpected token '{token}'";
      ER_2 = "Name or number expected, but '{token}' found";
      ER_3 = "Unknown variable '{name}'";
      ER_4 = "Dividing by zero";
}
```

Реализация пользовательских семантических процедур:

```
#ifndef CALC_H
#define CALC_H
```

```
#pragma once

#include "calc_impl.h"
#include "calc_impl_lex.h"

namespace ecg {

class CALC : public HELPER<CALC, CALC_IMPL, CALC_IMPL_LEX>
{
public:
	typedef const std::string & STRING;
	typedef P<int> PINT;

	void Assign(STRING name, PINT value, PINT & res)
	{
		vars[name] = *value;
		res = value;
		std::cout << "=(" << name << ") ";
	}

	void LINE(void)
	{
		std::cout << Line() << ": ";
	}

	void Print(PINT value)
	{
		std::cout << "-> " << *value << std::endl;
		for (std::map<std::string, int>::const_iterator
			iter = vars.begin(); iter != vars.end(); ++iter
		) {
			std::cout << "\t" << iter->first << "\t= "
				<< iter->second << std::endl;
		}
	}

	enum CALC_TYPE {
		ADD, SUB, MUL, DIV,
	};

	void Calc(CALC_TYPE type, PINT a, PINT b, PINT & res)
	{
		res = new int();
		switch (type) {
			case ADD: *res = *a + *b; std::cout << "+ "; break;
			case SUB: *res = *a - *b; std::cout << "- "; break;
			case MUL: *res = *a * *b; std::cout << "* "; break;
			case DIV:
				// ошибка при делении на ноль
				if (*b == 0) {
					PrintMessage("ER_4");
					StateCleanUp();
					return ;
				}
				*res = *a / *b; std::cout << "/ ";
				break;
		}
	}

	void Negate(PINT a, PINT & res)
	{
		res = new int(-*a);
		std::cout << "neg ";
	}
```

```
	enum VALUE_TYPE {
		NUMBER, NAME,
	};
	void Value(VALUE_TYPE type, STRING val, PINT & res)
	{
		std::cout << val << " ";

		if (type == NUMBER) {
			res = new int(atoi(val.c_str()));
			return ;
		}

		// ошибка, если пытаются получить значение неизвестной переменной
		if (vars.find(val) == vars.end()) {
			messageTable["name"] = val;
			PrintMessage("ER_3");
			StateCleanUp();
			return ;
		}

		res = new int(vars[val]);
	}

	// функция вывода сообщений
	void PrintMessage(STRING msgId)
	{
		messageTable["token"] = TokenValue();
		string format = GetMessage(msgId)->format;
		cout << endl << Compose(format, messageTable) << endl;
	}

	// сообщение об ошибке для правила-обработчика ошибок
	void Message(STRING msgId, STRING name, PINT & res)
	{
		if (IsCleanUp()) {
			return ;
		}

		PrintMessage(msgId);
		StateCleanUp();
	}

	// общий метод вывода ошибок
	void		Error		(int state)
	{
		PrintMessage("ER_1");
		StateCleanUp();
	}

private:
	std::map<std::string, int> vars;
};

}

#endif // CALC_H
```

Если подать данной реализации на вход следующий текст с ошибками (ошибки выделены красным цветом):

```
a = -5;
inf = 8 / 0 *;
b = +a + 5 * a * (a - 3);
c = a * 5 - -3 * b;
a = b = c;
some1 = 12 * 6 - 3 / 2 ;
b = (c = some1 + a * 3) / 5;
```

то будет распечатано (сообщения об ошибках выделены другим цветом):

```
2: 5 neg =(a) -> -5
        a       = -5
3: 8 0
Dividing by zero
4:
Name or number expected, but '+' found
5: a 5 * 3 neg b
Unknown variable 'b'
6: c
Unknown variable 'c'
7: 12 6 * 3 2 / - =(some1) -> 71
        a       = -5
        some1   = 71
8: some1 a 3 * + =(c) 5 / =(b) -> 11
        a       = -5
        b       = 11
        c       = 56
        some1   = 71
```

Приложение C. Грамматика *ECG*

Ниже приведена грамматика парсера *ECG*. Для простоты, семантические правила опущены:

```
#name LOADER, LOADER_IMPL

#autotype

COMMA             = ","
LB                = "("
RB                = ")"

$word             = {\D^\y}{\S^\y}*
$literal          = "(\\{^\n}|{^\\\n\"})*"

cline             = //
cbegin            = /{*}
cend              = {*}/

skip              = \s

preset default {
    +all
}

preset eoln {
      skip        = {\s^\n}
      $eoln       = \n
      +all
}

preset preset {
      "_"
      +$word +$literal +"skip" +"{" +"}" +"->" +"+" +"="
}

preset regexp {
      $regexp     = \S+
      -comment
}

preset name {
      $name       = {\S^\y}+
}

$word, $literal, $name, $regexp : string

program           -> "#name" worl "," worl $eoln (declaration *);

declaration       -> "#start" worl $eoln;

worl              -> $word | $literal;
worl              -> "L" $literal;

declaration       -> (param % "|") code_block;

code_block        : CODE_BLOCK
code_block        -> "<<<" code_name ">>>";
code_block        -> "{"   code_name "}";
```

```
code_block        -> "<<<" code_name params_ne ">>>";
code_block        -> "{"   code_name params_ne "}";
code_name         -> worl | "construct";

params_ne         -> LB (param % COMMA) RB;

declaration       -> rule ";" attrib_list;
attrib_list       : RULE_ATTRIB
attrib_list       -> ;
attrib_list       -> attrib_list "shift" shift_params;
attrib_list       -> attrib_list "reduce" shift_params;
attrib_list       -> attrib_list "prior" "(" $word ")";
attrib_list       -> attrib_list code_block;

shift_params      -> [LB (shift_term % COMMA) RB];

shift_term        : ITEM
shift_term        -> $literal;
shift_term        -> $word;
shift_term        -> "$eof";

rule              : RULE
rule              -> $word {FNT} "->" rule_list (nt_type -> [":" type]);

rule_list         -> rule_item % "|";

rule_item         : RULE_ITEM
rule_item         -> ;
rule_item         -> item_list; shift("]")
rule_item         -> item_list "*";
rule_item         -> item_list "+";
rule_item         -> "[" item_list "]";
rule_item         -> item_list "%" item_list;

item_list         -> item +;

item              : ITEM
item              -> $literal;
item              -> $word;
item              -> "$error";

item              -> "(" rule ")";
item              -> "(" rule_list nt_type ")";

item              -> code_block;
item              -> "[" rule_item "]";

declaration       -> (item2 % ",") ":" type;

item2             : ITEM
item2             -> $word;
item2             -> $literal;

type              : TYPE
type              -> $word;
type              -> $literal;
type              -> $word "*";
type              -> $literal "*";

param             : PARAM
declaration       -> "left"   (param +) ";";
declaration       -> "right"  (param +) ";";

param             -> $word;
```

```
param             -> $literal;
param             -> "L" $literal;

declaration       -> alias_name       "=" param;
declaration       -> alias_name       "=" $regexp regexp_prior;
alias_name        -> param;

regexp_prior      -> [{SPP} ":" $name]; // low, normal, high

declaration       -> "skip"   {ESP} "=" $regexp;
declaration       -> "cline"  {ESP} "=" $regexp;
declaration       -> "cbegin" {ESP} "=" $regexp;
declaration       -> "cend"   {ESP} "=" $regexp;

declaration       -> "preset" $name {PTNM} "{" (pr_com *) {PTND} "}";

pr_com            -> "+" param;
pr_com            -> "-" param;
pr_com            -> "skip"   {ESP} "=" $regexp;
pr_com            -> param    {CPT} "=" $regexp regexp_prior;
pr_com            -> param;

declaration       -> "#autotype";
declaration       -> "msg" "{" (message *) "}";

message           -> $word "=" $literal ["," $literal] ";";
```

Ниже приведена грамматика парсера регулярных выражений *ECG:*

```
#name LEX, LEX_IMPL

SLASH       = "\\"

program     -> e;

$char       = {^}

preset default {
      +$char +"{" +"}" +"\\"
      "(" ")" "[" "]" "*" "+" "|"
}

preset set {
      +$char  +"{" +"}" +"\\"
      "^" "-"
}

preset slash {
      $char_sl = {^}
}

left "|";
left concat $char "\\" "{";
left "*" "+";
left "(" "[";

$char       : string
$char_sl    : string
e           : REGEXP

"{"               <<< SetPreset >>>
"}"               <<< DefPreset >>>
SLASH             <<< SlashPreset >>>
$char_sl          <<< RestorePreset >>>
```

```
e                -> $char;
e                -> SLASH $char_sl;
e                -> set2;

set2, set, s : CHARS
set2         -> "{" set "}";
set2         -> "{" set "^" set "}";
set              -> ;
set              -> set s;
s                -> $char;
s                -> $char "-" $char;
s                -> SLASH $char_sl;
s                -> set2;

e                -> e e;     prior(concat)
e                -> e "*";
e                -> e "+";
e                -> e "|" e;
e                -> "[" e "]";
e                -> "(" e ")";
```