Тез. докл. 2-ой Межд. конф. по проблемам управления. М., 2003. Т. 2. С. 165.

УДК 519.681

КОМПИЛЯЦИЯ ЗНАНИЙ, ПРЕДСТАВЛЕННЫХ НА ЯЗЫКЕ ESSE

В.С. Выхованец, В.Я. Иосенкин
Приднестровский государственный университет им. Т.Г. Шевченко
Молдова, 3300, Тирасполь, 25 Октября ул., 128
e-mail: slava@tiraet.com, vik@tirastel.md

Язык контекстного программирования Esse [1] предлагает широкие возможности в области представления знаний в текстовой фоме с использованием контекстной привязки терминов (в языке слова) исследуемой предметной области. В данной статье рассматриваются принципы и механизмы компиляции знаний, представленных на языке Esse, в код виртуальной машины с применением высокоуровневой формы представления мигрирующего кода [2]. Компилятор Esse сочетает в себе сразу ряд полезных черт: просмотр вперед для сопоставления правого контекста и проверки синтаксиса, расширенный перебор с возвратами, автоматическое создание локальных переменных, отсрочка компиляции кода до момента исполнения в ситуации, когда необходимую для порождения кода информацию трудно извлечь во время компиляции, но легко получить во время исполнения, нет ключевых слов (есть только один зарезервированный знак @ - запись вершины стека в область кода из стека операндов). Часть из них по отдельнолсти имеется в существующих на сегодняшний день компиляторах. Применение их в совокупности с контекстной интерпретацией слов, дает то, новое качество, которое необходимо для повышения надежности и ускорения разработки программ, - ликвидация семантического разрыва между математической моделью и языковыми средствами.

При компиляции контекстного языка возникает ряд интересных моментов, которые необходимо рассмотреть при реализации. Для универсальности и независимости языка необходимо решить вопрос с используемыми типами данных. Предлагается два решения:

1. Использование одного скалярного (текстового) типа данных, котрый будет по разному интерпретироваться виртуальной машиной в зависимости от конкретного содержания данных (использование полиморфизма комманд) или отдельно задаваемой для каждого типа данных команды (например, отдельная команда для сложения чисел и конкатенации строк). Данный подход будет весьма удобен при реализации виртуальной машины на языке Perl, который фактически работает только со строками.

   Описание литерала на языке Esse будет выглядеть в этом случае так:

   ```
   essence is essence:
       essence '@', [].
   literal is essence:
       '".+"', literal.
   ```

   Литерал – есть сущность, задаваемая шаблоном - все что угодно в одинарных кавычках. Далее содержимое ковычек при необходимости конвертируется в необходимый тип данных. Единственная операция, котрая применима для сущности (essence) - это операция @, т.е. запись литерала в область кода.

2. Использование различных типов данных с явным указанием их типа, рекомендуется при использовании для реализации виртуальной машины на C++ подобных языках.В этом случае рекомендуется определить следующие типы данных: slit – текстовый литерал, ilit – целочисленный литерал, flit – числовой литерал с плавающей точкой, blit – бинарный литерал для хранения любых данных (изображения, видео, музыка и .т.д.), представляющий из себы сторуктуру из двух полей, первое типа ilit, для хранения размера следующих во втором поле данных, и собственно этих даных. Для хранения информации о типе используется код типа – число, например, для slit – 0, ilit - 1, flit – 2, blit – 3.

   Однако использование ключевых слов, которые обычно используются для явного указания типов данных, в языке не допускается. Поэтому сначала декларируются типы данных, в том чиле и целочисленый тип данных, котрый и

используется для хранения кода типа, после чего код типа данных уже является не ключевым словом, а просто словом iliteral языка Esse:

```
literal is essence:
        '".+"', sliteral;
        "[0-9]+", iliteral.
…
        '".+"' [#0 @, 0 @], sliteral;
        "[0-9]+" [#0 @, 1 @], iliteral.
```

Также возникают вопросы и с приоритетом предложений, описывающих понятия [3]. Предлагается два решения:

1. Задавать приоритет порядком появления описания предложения в тексте программы. Вариант удобен своей наглядностью и простотой, но имеет недостаток – возникает вопрос, как быть в том случае, если необходимо, чтобы приоритет дополнительно орпеделенного предложения был выше, чем тот, который будет присвоен ему в порядке следования по тексту программы. В этом случае придется вводить дополнительную конструкцию для явного определения приоритета вновь определяемого предложения.
2. Задавать приоритет предложений через ввод нового понятия. На примере видно, как задается приоритет арифметический операций:

```
primary as numeric:
        numeric ['get' @], value;
        numeric '++', primary { %0 get dup inc %0 put };
        numeric '--', primary { %0 get dup dec %0 put };
        '++' numeric, primary { %0 get inc dup %0 put };
        '--' numeric, primary { %0 get dec dup %0 put };
        "([\-+]?[0-9]+)", primary;
        '(' integer ')', primary.
unary is primary :
        '-' primary, unary;
        '+' primary, unary;
        '~' primary, unary.
multiplication is unary :
        multiplication '*' unary, multiplication ;
        multiplication '/' unary, multiplication;
        multiplication '%' unary, multiplication.
addition is multiplication :
        addition '+' multiplication, addition;
        addition '-' multiplication, addition.
```

В данном случае приоритет задается косвенно путем формирования понятий, объединяющих предложения с одинаковым приоритетом, фактически происходит наследования понятий. Такой метод выглядит достаточно естественно с точки зрения мышления человека, единственный минус – необходимость вводить новые понятия.

В языке Esse используется задание приоритетов предложений с использованием ввода новых понятий для разбиения предложений с равными приоритетами, а также внутри каждого понятия приоритет внутри понятия задается порядком определения предложений (предложение, определенное последним имеет наивысший приоритет).

Для обработки структуры предложений [3, 4] необходимо использование трех массивов слов: контекст backward, правый контекст forward (начинается с первого терма встреченного в предложении из входного потока) и множество продукций produce. При обработке входного потока с массивом понятий в produce происходит могократный вызов копий компилятора для анализа образующегося дерева понятий, которое требует довольно сложной обработки. При изучении ряда предметных областей, выяснено что в подавляющем большинстве случаев нет необходимости в массиве продукций, поскольку достаточно всего одной. Это следует из образа мышления человека: при формировании понятий несколько понятий порождают только одно понятие, а не несколько. Например, яркая звезда, центр вращения планет, святящийся шар, небесное светило описыают одно понятие – Солнце. Более того, если даже возникает ситуация, когда необходимо порождать сразу несколько понятий, всегда можно ввести новое понятие, как

совокупность порождаемых понятий. И именно оно будет продукцией предложения. Данное усечение функциональности нисколько не уменьшает выразительной мощности языка, но значительно упрощает его реализацию и, поскольку язык одновременно интерпретируемый и компилируемый, ускоряет исполнение программ.

Лексический анализатор языка Esse реализуется предельно просто по причине отсутствия в языке ключевых слов. При чтении новых символов из входного потока есть необходимость контролировать только три входных символа – предыдущий, текущий и следующий для распознания правых и левых скобок и знаков пунктуации. В качестве токенов в языке выступают символы пунктуации (различные виды скобок “(”, “)”, “[”, “]”,“{”, “}”, “<”, “>” и “;” “:” “,” “.”) и строки литералов. В данном случае разумнее всего разрабатывать лексический анализатор в виде процедуры, вызываемой синтаксическим анализатором для получения очередного токена.

Синтаксический анализатор (парсер) языка Esse, строится на основе LL-граматики, приведенной в [3]. Там же описана работа анализатора, иерархия словарей и структура сущностей. Для разбора грамматики используется нисходящий анализ методом рекусивного спуска [5] с использованием иерархии сущностей в виде ассоциативного дерева понятий в качестве таблицы предиктивного анализа и алгоритма динамического программирования [6,7] с использованием откатов. Реализация синтаксического анализатора фактически представляет переход от высокоуровневой формы мигрирующего кода к компилируемым и одновременно интерпретируемым конструкциям compile и definition, преставляющих собой более низкоуровневый код с использованием, как ранее определенных высокоуровневых конструкций и понятий, так и низкоуровневых команд виртуальной машины. После любого слова предложения из входного потока могут встречаться конструкции compile и definition, предсталяющих собой структурированный текст - интрукции компилятора, задающие операции которые необходимо выполнить на этапе компиляции или исполнения кода. Соответственно имеются два режима работы генератора промежуточного кода (далее будем называть компилятор) – режим compile (при обработке текста в квадратных скобках “[“ “]”) и режим execute (при обработке текста в квадратных скобках “{“ “}”). Разделение режимов компиляции и исполнения позволяет добиться большей гибкости и эффективноти. Особенно это важно, когда для компиляции некого кода необходимо использовать информацию, доступную только на этапе исполнения (так на этапе компиляции не доступны значения локальных переменных), в этом случае компиляция этого кода может быть отложена до этапа исполнения (динамическая генерация кода). Компилируемый и исполняемый текст состоит из блоков, минимальной составной единицей которых является фраза, которые разделяются между собой “.” “;” или “,” . Именно фраза phrase и обрабатывается компилятором.

Перед входом в компилируемый или исполняемый текст происходит инициализация компилятора, - определяются системные переменные компилятора: текущее обрабатываемое понятие, текущее обрабатываемое предложение, область видимости, режим работы компилятора; очищается стек операндов (далее просто стек) и инициализируется область кода предложения. По выходу из текста осуществляется проверка на соответствие содержимого стека тому, что должно быть на его вершине в соответствие с множеством продукций, определенных в качестве результата исполнения текущего компилируемого предложения, а также заносится код завершения ('exit') в область его кода.

Фразы текста обрабатываются поочередно. В качестве точки входа в дерево словарей entry по умолчанию задается текущее обрабатываемое понятие. Обработка фразы происходит по одному токену, в случае необходимости производятся откаты назад. Для нахождения соответствия фразы какому-либо ранее определенному предложению происходит циклический перебор всех предложений понятия entry, если среди них не найдено подходящее, осуществляется переход к понятию родительскому entry и т.д. до корневого понятия. Обработка предложений происходит начиная от самого большего по номеру (т.е. определенного последним) и до нулевого (определенного первым) в соответствии с вышеописанным уровнем приоритетов. Для проверки соответствия фразы текста предложению, проверяется токен предложения на совпадение с токеном из входного потока, проверяется контекст предложения на соответствие содержимому на вершине стека. Далее компилятор заглядывает вперед с пустым стеком и анализирует правый контекст. Ищет сопоставление входного потока с конструкциями в словарях, которые дают требуемые понятия в стеке правого

контекста forward, которые сопоставляются с понятиями в определении текущего предложения после терминального знака. Проверяется, есть ли хоть одно понятие в стеке и стеке форвард сопоставленное друг другу, Например, в стеке есть conjunction, а в стеке форвард - boolean, тогда conjunction преобразуется в boolean, и тем самым происходит сопоставление понятий и удаляется conjunction из стека и boolean из стека forward - и процесс повторяется, пока не исчерапается стек. Сопоставление понятий осуществляется благодаря наследованию понятий через конструкции подобия "is" и "as". "is" реализуется непосредственно с помощью языковых конструкций, “as” – через механизмы конвертирования, задаваемые в предложениях в виде блоков в квадратных скобках “[“ “]”, описывающих, какие преобразования с родителем необходимо произвести, чтобы получить новое понятие.

Так в результате компиляции последнего предложения из ниже приведенного примера, создается переменная типа boolean через конструкцию 'local' name 'is' nation, где name – имя переменной, nation – тип, а 'local' и 'is' – токены предложения, необходимы для удобного представления данной операции. Всю низкоуровневую работу по процессу формирования локальной переменой выполняет команда виртуальной машины 'loc'.

```
local as name:
        'local' name 'is' nation ['loc' @];
        local 'get' ['get' @], value;
        value local 'put' ['put' @].
boolean follows.        // декларация определение понятия boolean без его описания.
logical is local:
        logical 'is' boolean ['swap' @, 'put' @];
        'boolean' name {local %0 is boolean}.
```

Если текущий токен является локальной переменной, она извлекается из входного потока и выполняются соответствующие действия (кладется ее значение в стек). Так %0 в последнем предложении примера заменяется на name (%0 - автоматически созданная локальная переменная или ссылка на понятие номер 0 в предложении).

Процесс поиска подходящего предложения идет до тех пор пока не встретиться не опознанный, как часть предложения, токен (“}” “,” “;” “.” “]” “)”). После того, как предложение, сопоставленное текущей обрабатываемой фразе найдено, из стека удаляется контекст ее применения, в область кода записывается необходимый код и на вершину стека кладется результат применения данного предложения из стека продукций предложения.

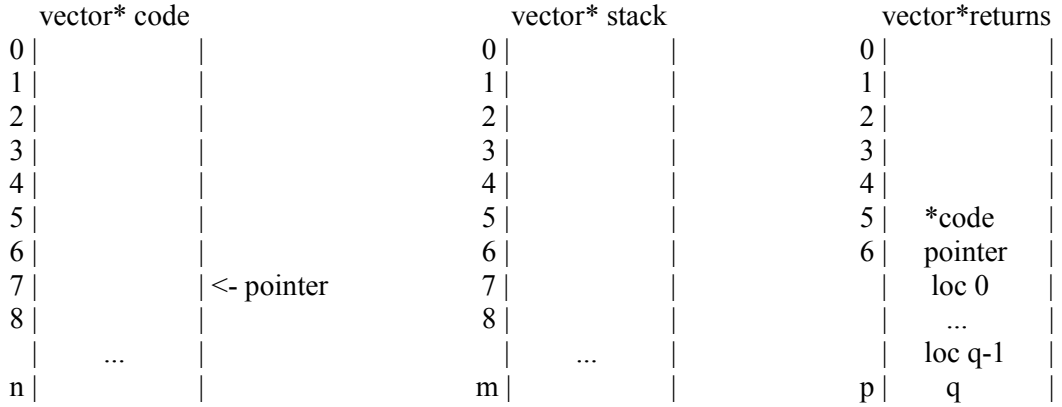
Ниже приведен текст дебугера компилятора Esse при компиляции фразы local %0 is boolean, на котором виден рекурсивный процесс ее обработки (в тексте дебугера сообщения поступают от модуля TOKEN – лексический анализатор, PARSER – синтаксический анализатор, SENTENCE – модуль обработки словаря, COMPILE – семантический анализатор и генератор кода виртуальной машины, RUN – виртуальная машина)

```
TOKEN:  New token is "loc"
  PARSER: Text of sentence in compile mode. Token "loc".
  SENTENCE: Clear code. Token "loc".
  SENTENCE: Put code ( 2 entry ) to ( ) Token "loc".
  COMPILE: +Begin compile local[ 0 ] in compile mode. Token "loc".
    PARSER: Block of text. Token "loc".
    PARSER: Narration of text. Token "loc".
    PARSER: Phrase of text. Token "loc".
      COMPILE: Start at 'local' in compile mode with stack ( ). Token "loc".

      COMPILE: Save (370 2 0). Token "loc".
      COMPILE: Look up '(local)'. Token "loc".
      COMPILE: Sentence [3] ( '%1' , nation ) skiping. Token "loc".
      COMPILE: Sentence [2] ( '$1' , essence ) skiping. Token "loc".
      COMPILE: Sentence [1] ( '%0' , name ) skiping. Token "loc".
      COMPILE: Sentence [0] ( '$0' , essence ) skiping. Token "loc".
      COMPILE: Look up 'local'. Token "loc".
      COMPILE: Sentence [0] ( 'local' name 'is' nation  ) skiping. Token "loc".
      COMPILE: Look up 'name'. Token "loc".
```

```
  COMPILE: Sentence [0] ( "[A-Za-z][0-9A-Za-z]*" , name  ) skiping. Token "loc".
  COMPILE: Look up 'literal'. Token "loc".
  COMPILE: Left 0 ( ) to context ( ). Token "loc".
  COMPILE: Sentence literal[0]: ( "'.+'" , literal  ). Token "loc".
  COMPILE: ( ) <--> ( "'.+'" [ 3 entry 1 get @ exit ] ). Token "loc".

  COMPILE: ( ) <--> ( [ 3 entry 1 get @ exit ] ). Token '@'.
  COMPILE: Execute code ( 3 entry 1 get @ exit ). Token '@'.
  RUN: Start [ 2 'loc' "loc" ], { }, ( 3 entry 1 get @ exit )
  RUN: Entry 3 parameners from stack to returns { 2 'loc' "loc" 3 }
  RUN: Put 2 to code at 'loc' ( 2 entry )
  RUN: Stop [ ], { }, ( )
  SENTENCE: Sentence opcode ( literal 0 ). Token '@'.
  COMPILE: Code 'literal',  '0'. Token '@'.
  COMPILE: Sentence match. Stack ( literal ). Token '@'.
PARSER: End sentence of phrase. Token '@'.
  COMPILE: Start at 'local' in compile mode with stack ( literal ). Token '@'.

  COMPILE: Save (375 2 1 literal    ). Token '@'.
  COMPILE: Look up '(local)'. Token '@'.
  COMPILE: Sentence [3] ( '%1' , nation ) skiping. Token '@'.
  COMPILE: Sentence [2] ( '$1' , essence ) skiping. Token '@'.
  COMPILE: Sentence [1] ( '%0' , name ) skiping. Token '@'.
  COMPILE: Sentence [0] ( '$0' , essence ) skiping. Token '@'.
  COMPILE: Look up 'local'. Token '@'.
  COMPILE: Sentence [0] ( 'local' name 'is' nation  ) skiping. Token '@'.
  COMPILE: Look up 'name'. Token '@'.
  COMPILE: Sentence [0] ( "[A-Za-z][0-9A-Za-z]*" , name  ) skiping. Token '@'.
  COMPILE: Look up 'literal'. Token '@'.
  COMPILE: Sentence [0] ( "'.+'" , literal  ) skiping. Token '@'.
  COMPILE: Look up 'essence'. Token '@'.
  COMPILE: Left 1 ( literal ) to context ( essence ). Token '@'.
  COMPILE: Sentence essence[0]: ( essence '@'  ). Token '@'.
  COMPILE: ( ) <--> ( '@' ). Token '@'.

  SENTENCE: Sentence opcode ( essence 0 ). Token ']'.
  COMPILE: Code 'essence',  '0'. Token ']'.
  SENTENCE: Put code ( @ ) to ( 2 entry ) Token ']'.
  COMPILE: Sentence match. Stack ( ). Token ']'.
PARSER: End sentence of phrase. Token ']'.
  COMPILE: Start at 'local' in compile mode with stack ( ). Token ']'.

  COMPILE: Save (377 3 0 ). Token ']'.
  COMPILE: Look up '(local)'. Token ']'.
  COMPILE: Sentence [3] ( '%1' , nation ) skiping. Token ']'.
  COMPILE: Sentence [2] ( '$1' , essence ) skiping. Token ']'.
  COMPILE: Sentence [1] ( '%0' , name ) skiping. Token ']'.
  COMPILE: Sentence [0] ( '$0' , essence ) skiping. Token ']'.
  COMPILE: Look up 'local'. Token ']'.
  COMPILE: Sentence [0] ( 'local' name 'is' nation  ) skiping. Token ']'.
  COMPILE: Look up 'name'. Token ']'.
  COMPILE: Sentence [0] ( "[A-Za-z][0-9A-Za-z]*" , name  ) skiping. Token ']'.
  COMPILE: Look up 'literal'. Token ']'.
  COMPILE: Sentence [0] ( "'.+'" , literal  ) skiping. Token ']'.
  COMPILE: Look up 'essence'. Token ']'.
  COMPILE: Sentence [0] ( essence '@'  ) skiping. Token ']'.
```

```
   COMPILE: Sentence no match. Stack ( ). Token ']'.
  COMPILE: Reduce ( ) to ( ). Token ']'.
  COMPILE: Convert ( ) to ( ). Token ']'.
  SENTENCE: Put code ( exit ) to ( 2 entry @ ) Token ']'.
 COMPILE: -End compile of 'local' in compile mode. Token ']'.
```

После обработки исполняемого текста компилятор осуществляет трансляцию входного потока в команды виртуальной стековой машины. Вируальная машина вызывается всякий раз, когда в forward встречается исполняемая конструкция. Для этого в нее передается код для исполнения и указатели на стек операндов и возвртов, их структура на рисунке ниже.



В стеке возвратов хранятся адреса на область кода и указатель на исполняемую команду, а также локальные переменные, на вершине стека хранится число локальных переменных до ближайшего указателя команд, это необходимо для определения глубины стека на которой сохраняется адрес возврата в область кода и указатель команд для реализации команд ветвления и переходов.

Ассоциативный массив мнемонических кодов базовых операций виртуальной машины языка Esse:

```
//      Compile
        com_code        =>      '@',
        com_pointer     =>      '$',
        com_nation      =>      '%',
        com_sentence    =>      '#',
//      Modules
        mod_entry       =>      'entry',
        mod_end         =>      'end',
        mod_exit        =>      'exit',
//      Branches
        bra_jmp         =>      'jmp',
        bra_jz          =>      'jz',
        bra_jnz         =>      'jnz',
        bra_js          =>      'js',
        bra_jns         =>      'jns',
//      Loclas
        loc_new         =>      'loc',
        loc_get         =>      'get',
        loc_put         =>      'put',
//      Globals
        glb_new         =>      'glb',
        glb_get         =>      'gget',
        glb_put         =>      'gput',
//      Stack
        stk_drop        =>      'drop',
        stk_dup         =>      'dup',
```

```
//      Logic
        log_not         =>      'not',
        log_or          =>      'or',
        log_and         =>      'and',
        log_xor         =>      'xor',
//      Arithmetics
        ari_neg         =>      'neg',
        ari_inc         =>      'inc',
        ari_dec         =>      'dec',
        ari_add         =>      'add',
        ari_sub         =>      'sub',
        ari_mul         =>      'mul',
        ari_div         =>      'div',
        ari_mod         =>      'mod'
```

После разбора определенных синтаксических конструкций запучкается виртуальная машина, которая последовательно обрабатывает весь полученный код, выполняя низкоуровневые инструкции. Например, при исполнении команды 'loc', выполняются следующие действия: берется значение nation с вершины стека stack, берется name с вершины стека stack, берется значение с вершины стека возвратов q и кладется в него пустое значение (значение локальной перемнной – пока пустое, его можно потом заменить на любое значение командой put) и q+1 (количество локальных переменных увеличилось на 1), создается новая локальная переменная для текущего предложения с именем name, адресом ее значения в стеке возвратов q, кладется в стек продукций переменной понятие nation.

Для исследования возможностей контекстной технологии программирования был разработан контекстный язык Esse, который был реализован авторами в виде компилятора-интерпретатора языка. В статье были рассмотрены модули компилятора. Синтаксический анализатор вызывает лексический анализатор, когда необходим новый токен, и вызывает семантическую процедуру, когда распознана конструция входного языка. Семантический анализатор создает внутренее представление язхыка в виде дерева сущностей и генерирует код для виртуальной машины. Компилятор был написан на языке С для интерпретации и компиляции контексного языка Esse, генерирующий код для виртуальной машины на внутреннем третьем языке, что характерно при использовании метода раскрутки [5]. В действительности на тексте языка Esse пользователем создается четвертый – прикладной язык, заданный в дереве сущностей, формируемого в декларативной части языка, – язык описания некой прикладной области, который по сути и будет являться истинно контекстным языком программирования-манипулирования знаниями этой области, поскольку грамматика Esse предусматривает задание всех токенов (терминалов) в контексте (может быть и пустым) их применимости с ранее определенными понятиями (нетерминалами).

Литература

1. *Иосенкин В.Я. Выхованец В.С.* Применение технологии контекстного программирования для решения больших прикладных задач // Труды международной конференции «Параллельные вычисления и задачи управления» (PACO’2001). М.: Институт проблем управления им. В.А. Трапезникова РАН, 2001. – С.(4)-121-139.
2. *Иосенкин В.Я., Выхованец В.С.* Технология контекстного программирования в телекоммуникационных системах // Труды второй международной научно-практической конференции «Современные информационные и электронные технологии». Одесса: Друк, 2001. - С.118-119.
3. *Иосенкин В.Я., Выхованец В.С.* Контекстная модель технологического процесса предприятия // Труды II международной конференции «Идентификация систем и задачи управления» (SICPRO’03). М.: Институт проблем управления им. В.А. Трапезникова РАН, 2003. - С.859-871.

Тез. докл. 2-ой Межд. конф. по проблемам управления. М., 2003. Т. 2. С. 165.

4. *Иосенкин В.Я.* Контекстная интерпретация лексем // Материалы международной научно-практической конференции «Информационные технологии в науке и образовании». Шахты: Изд-во ЮРГУЭС, 2001. – С.73-75.
5. *Ахо А.*, Сети Р. и др. Компиляторы: принципы, технологии и инструменты. – М. : Издательский дом «Вильямс», 2001. – С. 167-201, 677-687.
6. *Younger D.H* Recognition and parsing of context-free languages in time n^3 // Information and Control 10:2, 1967. - PP 189-208.
7. *Kasami T.* An efficient recognition and syntax analysis algorithm for context-free languages, AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, Mass, 1965.