25

1. Метод рекурсивного спуску

рекурсия — способ общего определения объекта или действия через себя, с использованием ранее заданных частных определений. Рекурсия используется, когда можно выделить самоподобие задачи.

**Метод рекурсивного спуска** - это один из методов определения принадлежности входной строки к некоторому [формальному языку](http://ru.wikipedia.org/wiki/%D0%A4%D0%BE%D1%80%D0%BC%D0%B0%D0%BB%D1%8C%D0%BD%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA), описанному LL(k) [контекстно-свободной грамматикой](http://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BD%D1%82%D0%B5%D0%BA%D1%81%D1%82%D0%BD%D0%BE-%D1%81%D0%B2%D0%BE%D0%B1%D0%BE%D0%B4%D0%BD%D0%B0%D1%8F_%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)

**Идея метода**

Для каждого нетерминального символа *K* строится функция, которая для любого входного слова *x* делает 2 вещи:

* Находит наибольшее начало *z* слова *x*, способное быть началом выводимого из *K* слова
* Определяет, является ли начало *z* выводимым из *K*

Такая функция должна удовлетворять следующим критериям:

* считывать из еще необработанного входного потока максимальное начало *A*, являющегося началом некоторого слова, выводимого из *K*
* определять является ли *A* выводимым из *K* или просто невыводимым началом выводимого из *K* слова

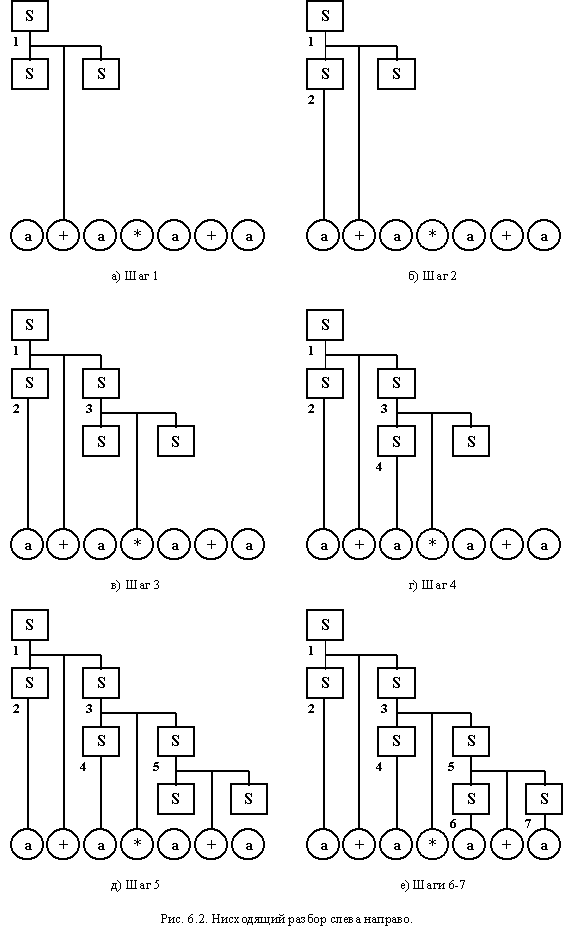
В случае, если такое начало считать не удается (и корректность функции для нетерминала *K* доказана), то входные данные не соответствуют языку, и следует остановить разбор.

Разбор заключается в вызове описанных выше функций. Если для считанного нетерминала есть составное правило, то при его разборе будут вызваны другие функции для разбора входящих в него терминалов. Дерево вызовов, начиная с самой “верхней” функции эквивалентно дереву разбора.

**Условия применения**

Пусть в данной формальной грамматике *N* - это конечное [множество](http://ru.wikipedia.org/wiki/%D0%9C%D0%BD%D0%BE%D0%B6%D0%B5%D1%81%D1%82%D0%B2%D0%BE) нетерминальных символов; *Σ* - конечное множество терминальных символов, тогда метод рекурсивного спуска применим только, если каждое правило этой грамматики имеет следующий вид:

* или A \rightarrow \alpha, где \alpha \subset (\Sigma \cup N)*, и это единственное правило вывода для этого нетерминала
* или A \rightarrow a_1\alpha_1|a_2\alpha_2|...|a_n\alpha_nдля всех i=1,2...n; a_i \ne a_j, i \ne j; \alpha \subset (\Sigma \cup N)*

*Метод рекурсивного спуска* учитывает особенности современных языков программирования, в которых реализован рекурсивный вызов процедур. Если обратиться к дереву нисходящего разбора слева направо то можно заметить, что в начале происходит анализ всех поддеревьев, принадлежащих самому левому нетерминалу. Затем, когда самым левым становится другой нетерминал, то анализ происходит для него. При этом используются и полностью раскрываются все нижележащие правила. Это навязывает определенные ассоциации с иерархическим вызовом процедур в программе, следующих друг за другом в охватывающей их процедуре. Поэтому, все нетерминалы можно заменить соответствующими им процедурами или функциями, внутри которых вызовы других процедур - нетерминалов и проверки терминальных символов будут происходить в последовательности, соответствующей их расположению в правилах. Такая возможность подкрепляется и другой ассоциацией. Вызов процедуры или функции реализуется через занесение локальных данных в стек, который поддерживается системными средствами. Заносимые данные определяют состояние обрабатываемого нетерминала, а машинный стек соответствует магазину автомата. *Использование рекурсивного спуска позволяет достаточно быстро и наглядно писать программу распознавателя на основе имеющейся грамматики*. Главное, чтобы последняя соответствовала требуемому виду. Использование рекурсивного спуска позволяет написать программу быстрее, так как не надо строить автомат. Ее текст может быть и менее ступенчатым, если использовать инверсные условия проверки или другие методы компоновки текста. Не имеет смысла заменять рекурсивный спуск автоматом с магазинной памятью, особенно в том случае, если грамматику задавать с использованием диаграмм Вирта.

<ВІДПОВІДЬ ПРО РЕКУРСИВНИЙ СПУСК, НЕ ЗВ’ЯЗАНА ІЗ СИНТАКСИЧНИМ РОЗБОРОМ:>

Порождение все новых копий рекурсивной подпрограммы до выхода на граничное условие называется *рекурсивным спуском*. Максимальное количество копий рекурсивной подпрограммы, которое одновренно может находиться в памяти компьютера, называется *глубиной рекурсии*. Завершение работы рекурсивных подпрограмм, вплоть до самой первой, инициировавшей рекурсивные вызовы, называется *рекурсивным подъёмом*.

Классический пример рекурсии — определение факториала. С одной стороны, факториал определяется так: *n*!=1\*2\*3\*...\**n*. С другой стороны, Факториал, рекурсивное определениеГраничным условием в данном случае является *n*<=1.

/\* Функция на C \*/ // Рекурсивный спуск – повторный вызов метода Factorial

double Factorial(int N)

{

double F;

if (N<=1) F=1.; else F=Factorial(N-1)\*N;

return F;

}.

2. Розширення MMX

MMX (Multimedia Extensions — мультимедийные расширения) — коммерческое название дополнительного набора инструкций, выполняющих характерные для процессов кодирования/декодирования потоковых аудио/видео данных действия за одну машинную инструкцию. Впервые появился в процессорах Pentium MMX. Разработан в лаборатории Intel в Хайфе, Израиль, в первой половине 1990-х.

::: Основа аппаратной компоненты расширения mmx – *восемь* новых *регистров*, которые на самом деле являются регистрами *сопроцессора*, только вместо 80-ти разрядов *используется 64 младших разряда* (мантисса). при работе со стеком сопроцессора в режиме mmx он рассматривается как обычный массив регистров с произвольным доступом. использовать стек сопроцессора по его прямому назначению и как регистры mmx-расширения одновременно невозможно. Основным принципом работы команд mmx является одновременная обработка нескольких единиц однотипных данных одной командой.

Допустим, у нас есть два массива: A и B, длиной по 8 байтов каждый. Поставим себе задачу — прибавить к каждому элементу массива B соответствующий ему элемент массива A. Это приведет нас к такой программе:

mov esi, offset A

mov edi, offset B

mov ecx, 8

OurLoop:

mov al, [esi]

add [edi], al

inc esi

inc edi

loop OurLoop

Как видим, не смотря на простую постановку задачи, процессор будет вынужден выполнить восемь раз нетривиальную последовательность команд. Вот если бы операцию сложения двух массивов можно было выполнить одной командой... На этой идее и был построен MMX. Аналогичная программа с его использованием имела бы такой вид: movq mmreg1, A

movq mmreg2, B

paddb mmreg2, mmreg1

movq B, mmreq2

Допустим, выполняется программа обрабатывающая 256-цветное изображение. Обычно она обрабатывает каждый пиксел по отдельности, используя же MMX, она может обрабатывать восемь пикселей сразу! Так что же такое MMX? MMX - это расширение, включающее в себя 57 новых команд и восемь 64-разрядных регистров. В основу положен принцип SIMD (Single Instruction Multiple Data) - одна инструкция - множество данных. MMX предоставляет инструкции для складывания, умножения и даже выполнения комбинированных операций. К примеру, команда PMADDWD перемножает, а затем складывает четыре слова данных, при этом она выполняется намного быстрее программы использующей только набор "стандартных" инструкций. Технология MMX предоставляет простую, гибкую программную модель, не требуя при этом перевода процессора в какой-то особенный режим. Все существующие программы будут корректно исполняться на процессорах с MMX без каких-либо изменений, даже если в системе присутствуют приложения, использующие новую технологию.

Регистры.

В процессорах использующих MMX добавлено 8 новых 64-разрядных регистров MM0-MM7. Они могут быть использованы только для выполнения операций с типами данных MMX. Команды MMX позволяют задавать в качестве операндов как регистры общего назначения (EAX, EBX, ECX, EDX, EBP, ESI, EDI и ESP), так и переменные в памяти, используя для этого стандартную схему адресации принятую в процессорах x86. Хотя MMX регистры и имеют ни с чем не совпадающие названия, на самом деле, они являются "псевдонимами" регистров сопроцессора (st0-st7). Это означает, что, изменяя один из регистров MMX, в своей программе, мы в то же время изменяем регистры сопроцессора.

**ТИПЫ ДАННЫХ**.

MMX поддерживает данные в упакованном формате. Это означает, что каждый 64 битовый регистр MMX может интерпретироваться как:

1. Восемь байтов

2. Четыре слова

3. Два двойных слова

4. Одна 64-разрядная переменная

Все инструкции можно разбить на следующие **группы**:

арифметические команды

сложение PADDB PADDD PADDSB PADDSW PADDUSB PADDUSW PADDW

сложение и умножение PMADDWD

умножение PMULHW PMULLW

вычитание PSUBB PSUBD PSUBSB PSUBSW PSUBUSB PSUBUSW PSUBW

команды очищающие регистры

EMMS

команды сравнения

проверка равенства PCMPEQB PCMPEQD PCMPEQW

сравнение PCMPGTB PCMPGTD PCMPGTW

команды упаковки/распаковки

PUNPCKHBW PUNPCKHDQ PUNPCKHWD PUNPCKLBW PUNPCKLDQ PUNPCKLWD PACKSSDW PACKSSWB PACKUSWB

логические команды

PAND PANDN POR PXOR

команды передачи данных

MOVD MOVQ

команды сдвига

логический сдвиг влево PSLLD PSLLQ PSLLW

арифметический сдвиг вправо PSRAD PSRAW

логический сдвиг вправо PSRLD PSRLQ PSRLW

Команды MMX имеют такой формат: инструкция mmreg1, mmreg2/mem64

То есть источником может быть как переменная памяти, так и регистр MMX. А целевым может быть только регистр MMX. К тому же MOVD и MOVQ допускают пересылки из регистров MMX в память. Одним из первых вопросов у меня был: "зачем так много команд?". Ведь количество операций не так уж и велико... Дело в том, что название команды формируется из двух частей. Первая часть говорит о том, что она делает (MOV,PSUB,PADD). Вторая же говорит о том, как она интерпретирует операнды. Рассмотрим команду PACKUSWB. Первая часть строки - PACK указывает, что будем что-то упаковывать. Вторая же - USWB в свою очередь разбивается на две US и WB. US говорит о том, что результат будет беззнаковым с сатурацией , WB же указывает, что источник - упакованные слова, результат - упакованные байты. Благодаря такой записи смысл инструкции схватывается "на лету", по этой же причине команд так много. Приведем полный перечень мнемоник:

P-упакованные данные (Packed data)

B-байт (byte)

W-слово (word)

D-двойное слово (DoubleWord)

Q-64 битовая переменная (QuadWord)

S-знаковая (Signed)

U-беззнаковая (Unsigned)

SS-знаковая с сатурацией (Signed Saturation)

US-беззнаковая с сатурацией (Unsigned Saturation)

Однако разрядность MMX регистров, к сожалению, мала. Поэтому некоторые классы задач сложно приспособить к использованию этой мощи. К примеру, в графике часто нужно выполнить операцию умножения матрицы на матрицу, или вектора на матрицу. Казалось бы, вот где MMX понадобится. Однако в большинстве задач в качестве элементов матрицы или вектора используются числа с плавающей точкой. Вывод - MMX тут не применишь.

Правда можно в качестве элементов использовать числа с фиксированной точкой . Размер такого числа - четыре байта. То есть в один регистр поместится только два элемента вектора. Тонкость тут в том, что в случае трехмерных преобразований скалярное произведение нужно считать от двух трехэлементных векторов... Решить проблему можно - использовать для этого два MMX регистра: в первом два элемента, во втором один. Однако выигрыш при этом будет не столь значителен, как хотелось бы.

КРАТКОЕ ОПИСАНИЕ КОМАНД.

**MOVD**

Формат

MOVD mmreg1, reg32/mem32

MOVD reg32/mem32, mmreg1

Описание

Инструкция MOVD пересылает 32 младших бита регистра MMX в регистр общего назначения или память. Или же из регистра общего назначения/памяти в регистр MMX. В последнем случае кроме собственно пересылки биты 32-64 соответствующего регистра обнуляются.

**PADDB**

Формат

PADDB mmreg1, mmreg2/mem64

Описание

Складывает восемь 8-разрядных чисел операнда источника (MMX регистра или 64-разрядного поля памяти) и восемь 8-разрядных чисел операнда-получателя (регистра MMX). В случае переполнения полученные значения заворачиваются (то есть 255+10=9) без установки флага переноса при подсчете последующих байтов. Результат помещается в операнд-получатель.

**PADDSW**

Формат

PADDSW mmreg1, mmreg2/mem64

Описание

Складывает четыре 16-битных чисел со знаком операнда источника (регистра MMX или 64-разрядного поля памяти) и четыре соответствующих значения операнда-получателя (регистра MMX). Если сумма каких-либо двух из них менее чем -32768 (8000h), то результат сложения -32768 (8000h). Аналогично в случае, когда сумма превышает 32767 (7FFFh), то возвращаемый результат 32767 (7FFFh). Результат заносится в операнд-получатель.

**PAND**

Формат

PAND mmreg1, mmreg2/mem64

Описание

Инструкция PAND выполняет логическую операцию "и" над операндами. Результат заносится в операнд-получатель. Если соответствующие биты источника и получателя равны единице, то значение итогового бита единица. Эта команда может использоваться для распаковки упакованных переменных при помощи маски полученной от инструкций сравнения PCMPEQ и PCMPGT.

**PCMPEQB**

Формат

PCMPEQB mmreg1, mmreg2/mem64

Описание

Сравнивает источник с получателем, рассматривая операнды как упакованные байты. Если биты операндов эквивалентны, то все биты 8-разрядной части получателя устанавливаются равными единице, в противном случае они обнуляются.

**PCMPGTD**

Формат

PCMPGTB mmreg1, mmreg2/mem64

Описание

Аналогична PCMPEQ, но в отличие от нее биты в целевом операнде устанавливается в том случае, когда байт целевого операнда больше соответствующего байта операнда источника.

**PACKUSWB**

Формат

PACKUSWB mmreg1, mmreg2/mem64

Описание

Переводит восемь знаковых слов задаваемые аргументами (по четыре слова в каждом) в восемь беззнаковых байт. После чего результат сохраняется в mmreg1

3. Організація підсистем введення – виведення

Початковим поштовхом до розробки ОС були проблеми автоматизації завантаження програм та використання узагальнених механізмів введення-виведення. На початковому етапі розроблення ОС найбільш дорогою частиною був ЦП, тому головною вважалася задача ефективного використання процесору Для цього основною проблемою була організація ефективного введення-виведення з одночасною роботою ЦП над розв’язанням інших задач. Для цього необхідно механізм апаратного переривання, який замінював цикл ЦП з очікуванням готовності даних.

Структура підпрограми драйверів пристроїв включала наступні блоки:

 Видача команди на пристрій для його підготовки до обміну

 Очікування готовності пристрою до обміну

 Виконання власне обміну

 Видача на пристрій команди для закінчення операції

 Організація обміну драйвера даними з програмою, яка його використовує.

При створенні мікропроцесорів фактично було повторено процес створення програм обміну для зовнішніх пристроїв, але з деякою систематизацією.

Підключення зовнішніх пристроїв до мікропроцесору виконувалось шляхом визначення вихідного командного порту, вхідного порту стану, які мали однакові номер та вхідного і вихідного порту для введення і виведення даних, які мали однакову адресу. Щоб написати узагальнений драйвер введення-виведення треба визначити адреси портів за допомогою

CMPRT EQU 41H

STPRT EQU CMPRT

INPRT EQU 42H

OUTPRT EQU INPRT

Возвращает в ax 1 байт данных введенных с некоторого устройства

drIn Proc

mov al,cmOn

out CMPRT,al

lwr: in al,STPRT

test al,RdyBit

jnz lwr

in al,INPRT

push ax

mov al,cmOff

out CMPRT

pop ax

ret

drIn endp