**SİSTEM PROGRAMLAMA**

**VE**

**İLERİ C UYGULAMALARI – II**

**Kursun Başlama Tarihi: Mayıs-2016**

**Eğitmen: Kaan ASLAN**

**1. GUI Ortamlarında Kullanılan Mesaj Tabanlı Programlama Modelinin Temelleri**

Bu bölümde bugün yaygın olarak kullanılan Windows gibi, XWindow gibi GUI ortamlarındaki mesaj tabanlı programlama modeli üzerinde durulacaktır. Bu bölümde kurs katılımcılarınına pencereli GUI (Graphical User Interface) sistemlerinde uygulama geliştirme becerisi kazandırma hedeflenmemektedir. Biz bu bölümde biz yalnızca sistem programcısı olarak mesaj tabanlı programlama modelinin alt yapısını aşağı seviyeli olarak ele almayı hedefliyoruz.

**1.1. Grafik Kartları ve Ekran Belleği**

Bilgisayar sistemlerine terminal bağlanması ilk kez 1957 yılında gerçekleştirilmiştir. Bu yıllar aynı zamanda programların zaman paylaşımlı (time sharing) olarak çalıştırılmaya başlandığı yıllardır. O zamana kadar insanlarlar bilgisayarlar doğrudan etkileşemiyordu. Programcı programını delikli kartlara delip bilgisayar operatörüne teslim ediyordu. Operatör de programı çalıştırıp sonucu programcıya kağıt çıktısı olarak veriyordu.

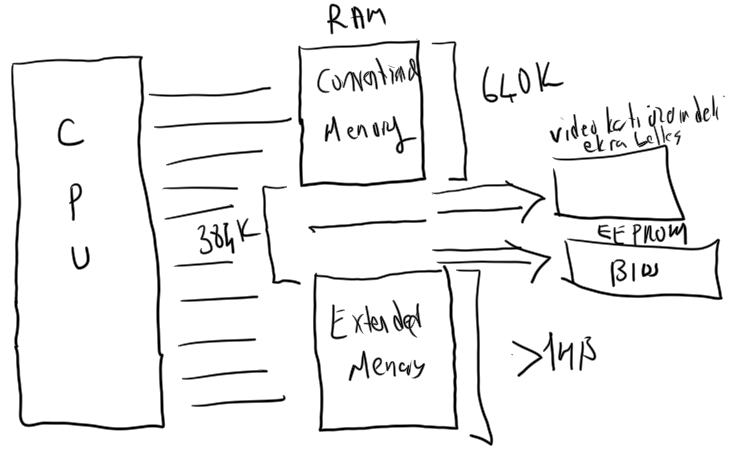
90’lı yılların ortalarına kadar terminal çalışması ağırlıklı olarak “text mode”da devam etmiştir. “Text mode” o zamanlar default çalışma moduydu. Bu çalışma modunda terminal ekranı yalnızca daktilodaki gibi karakterleri bir kalıp olarak ekrana basabiliyordu. Yani “text mode”da ekrana basılacak en küçük birim bir karakterdi.

İlk grafik ekranlar 70 yıllarda ortaya çıkmaya başladıysa da grafik modun default duruma gelmesi 90’lı yıllarda olmuştur. Grafik modda ekrana çıkartılabilecek en küçük birime pixel (picture element sözcüklerinden kısaltma) denilmektedir. Ekran bu modda pixel’lerden oluşan bir matris gibidir. Her pixe’lin rengi diğerlerinden bağımsız olarak atanabilmektedir. Böylece grafik ekranda aslında bütün görüntüler pixel’lerin yan yana getirilmesiyle oluşturulmaktadır.

Pekiyi neden grafik çalışma nispeten daha sonraları başlamıştır? İşte grafik ekrandaki pixel’lerin görüntülenmesi ve bunların grafik kartında oluşturulması için belli bir teknoloji düzeyi gerekiyordu. Bugün artık hem ekranlar hem de grafik kartları teknolojik olarak çok ilerlemiştir. Örneğin 1980-1981 yıllarına IBM ilk kişisel bilgisayarı (IBM PC) piyasaya sürdüğünde oradaki ekran ancak kalıp karakterlkeri görüntüleme kapasitesine sahipti. Yani “text mode”da çalışıyordu. IBM daha sonra sırasıyla EGA, VGA ve SVGA (Super VGA) kartlarını geliştirdi. Bu kartlar pixel düzeyinde grafik işlem yeteneğine sahipti. Tabii EGA ve VGA kartları grafik görüntüyü gösterebilen monitörlerle kullanılabiliyordu. 80’li yıllarda ayrıca Hercules kartları da yoğun olarak kullanılmıştır.

Bugün kullandığımız PC’lerde (PC’yi burada genel bir terim olarak kullanıyoruz. Bu terim notebook’ları ve diziüstü bilgisayarları da kapsamaktadır) ekran pasif bir birimdir. Görüntü bilgisayarın içerisindeki grafik kartının belleği üzerinde oluşturulur. Grafik kartı da belli bir periyotta (buna “refresh rate” deniyor) bu belleğin içeriğini ekrana yollar. Grafik kartı üzerindeki görüntünün oluşturulduğu bu belleğe “ekran belleği (video RAM)” denilmektedir. Ekran belleği doğrudan CPU’nun adres alanı içerisindedir. Bugün kullandığımız PC’lerde ilk 1 MB’nin son 384 K’sı BIOS’un içinde bulunduğu EEPROM belleğe ve ekran belleğine yönlendirilmiş durumdadır.

**Anahtar Notlar:** 8086 işlemcisiyle oluşturulmuş ilk PC’ler 1 MB belleği adresleyebiliyordu. Bu 1MB alanın son 384K’sı BIOS ve ekran belleğine yönlendirildiği için DOS işletim sistemi yalnızca bu 1 MB’nin 640K’sını kullanabiliyordu. Bu 640K’ya “geleneksel bellek (conventional memory)” deniliyordu.



Görüldüğü gibi RAM’in ilk 1MB’sinin son 384K’sında aslında RAM değil başka birimler bulunmaktadır. CPU orayı adreslerken orada RAM’in mi yoksa ekran belleğinin mi bulunduğunu bilmez. Eletrisksel olarak CPU’nun adres alanına farklı RAM blokalrı ve ROM’lar ya da başka aygıtlar bağlanabilir. Ekran belleği A0000 adresinden başlamaktadır. Biz C’de ya da sembolik makine dilinde oraya erişebiliriz. Ancak Windows gibi, Linux gibi işletim sistemlerinde ekran belleği korunmuştur. User mode prosesler oraya doğrudan erişemezler.

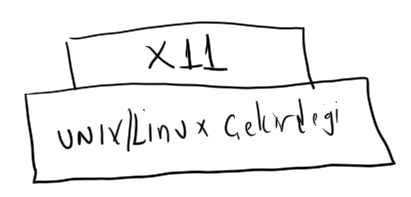
Bugün SVGA kartları çok gelişmiştir. Ekran kartının üzerindeki RAM’in bir bölümü ekran belleği için bir bölümü ise GPU’nun (Graphics Processing Unit) grafik işlemleri yapabilmesi için kullanılmaktadır. Bugünkü GPU’lar görüntü işleme konusunda rutinleri olan pek çok görüntü işlemini birinci elden yapabilen yetenekli işlemcilerdir. Örneğin bir şekil döndürmesini doğrudan GPU yapabilmektedir. Sonuç görüntüyü ekran belleğine aktarabilmektedir. Hatta GPU’lar paralel programlamada ayrı bir işlemci gibi bile bazı işlemlerde kullanılabilmektedir. Bugünkü ekran kartları ve GPU’lar bazı ayrıntılı özelliklere sahiptir. Kursumuzda bu ayrıntılardan bahsedilmeyecektir.

**Anahtar Notlar:** Genel olarak bilgisayar mimarisinde birtakım donanım aygıtlarının sanki RAM’in bir parçasıymış gibi gösterilmesine “memory mapped IO” denilmektedir. Grafik kartları tipik olarak “memory mapped IO”ya bir örnek oluşturmaktadır.

**1.2. GUI Sistemlerine Özet Bir Bakış**

Windows’un çekirdek (kernel) ile entegre edilmiş bir GUI alt sistemi vardır. Başka bir deyişle Windows’ta pencereli çalışma başka bir katman tarafından değil doğrudan işletim sisteminin bir parçası tarafından sağlanmaktadır. Windows’u GUI alt sistemi olmadan kullanmak mümkün olsa da anlamlı değildir.

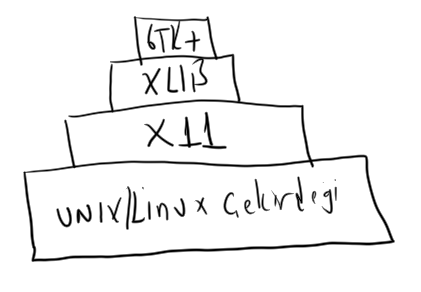
UNIX/Linux sistemlerinde grafik arayüz çekirdeğin üzerine oturtulan ve ismine X11 (ya da XWindow) denilen bir alt sistem tarafından sağlanmaktadır. Yani örneğin Linux’un çekirdeğinin kaynak kodlarında pencere kavramına ilişkin hiçbir şey yoktur. Ancak Windows’ta vardır.



**Anahtar Notlar:** Biz istersek bir Linux sistemini grafik arayüz olmadanda başlatabiliriz. Bunun için yapılması gereken tek şey “run level denilen bir değeri değiştirmektir. Bu işlem "/etc/init.d" ya da "/etc/inittab" dosyaları edit edilerek değiştirilebilir. (Örneğin "run level" değerini 3’e çekerek sistemin X11 olmadan başlatılmasını sağlayabiliriz.)

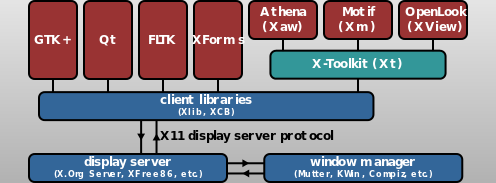
X11 grafik sistemi client-server tarzda çalışmaktadır. Yani sanki X11 bir server program gibi, pencere açmak ve pencereler üzerinde işlem yapmak isteyen programlar da client programlar gibidir. X11 sisteminde işlem yapabilmek için oluşturulmuş temel kütüphaneye XLIB denilmektedir. XLIB'i X11’in API kütüphanesi olarak düşünebiliriz. Son yıllarda XLIB’in XCB isimli daha modern bir versiyonu da oluşturulmuştur. XLIB ve XCB temelde C programlama dilinden kullanılmak için tasarlanmıştır. Ancak bu kütüphaneler başka dilelrden de kullanılabilmektedir.

Pek çok programcı için X11 sistemleri için tasarlanmış olan XLIB ve XCB oldukça aşağı seviyelidir. Bu kütüphanelerle kullanıcı arayüzü oluşturmak biraz zahmetlidir. İşte bu nedenle XLIB'den faydalanılarak daha yüksek seviyeli kütüphaneler de oluşturulmuştur. Bunların en yaygını GTK+’tır.



GTK+ da temelde bir C kütüphanesidir. GNOME pencere yöneticisi GTK+ kullanılarak yazılmıştır. GTK+'ın dışında ayrıca XLIB üzerine oturtulmuş Xt gibi, Motif gibi başka kütüphaneler de bulunmaktadır.

X11 sistemlerindeki kütüphaneler aşağıdaki şekille özetlenebilir:



Bu şekilden de gördüğünüz gibi Qt de tıpkı GTK+ gibi XLIB kullanılarak yazılmış bir framework ve kütüphanedir. Ancak GTK+ C Programlama Dilini için tasarlanmış olmasına karşın, Qt nesne yönelimli olarak C++ için tasarlanmıştır. Ayrıca Qt “cross platform” olmasından dolayı da pek çok programcı tarafından tercih edilmektedir.

**1.3. GUI Ortamlarında Mesaj Tabanlı Programlama Modeli**

Grafik arayüze sahip pencereli sistemlerde genel olarak mesaj tabanlı (message driven/event driven) çalışma modeli kullanılmaktadır. Mesaj tabanlı çalışma modelinin ayrıntıları sistemden sisteme değişebilmekle birlikte burada biz her sistemde geçerli olan bazı temel bilgileri vermekle yetineceğiz.

Mesaj tabanlı programlama modelinde klavye ve fare gibi aygıtlarda oluşan girdileri programcı kendisi almaya çalışmaz. Fare gibi, klavye gibi girdi aygıtlarını işletim sisteminin (ya da GUI alt sistemin) kendisi izler. Oluşan girdi olayı hangi pencereye ilişkinse işletim sistemi ya da GUI alt sistem, bu girdi olayını “mesaj” adı altında bir yapıya dönüştürerek o pencerenin ilişkin olduğu (yani o pencereyi yaratan) programın “mesaj kuyruğu (message queue)” denilen bir kuyuk sistemine yerleştirir. Mesaj kuyruğu içerisinde mesajların bulunduğu FIFO prensibiyle çalışan bir kuyruk veri yapısıdır. Sistemin daha iyi anlaşılması için süreci maddeler halinde özetlemek istiyoruz:

1. Her programın (ya da thread’in) “mesaj kuyruğu” denilen bir kuyruk veri yapısı vardır. Mesaj kuyruğu mesajlardan oluşmaktadır.

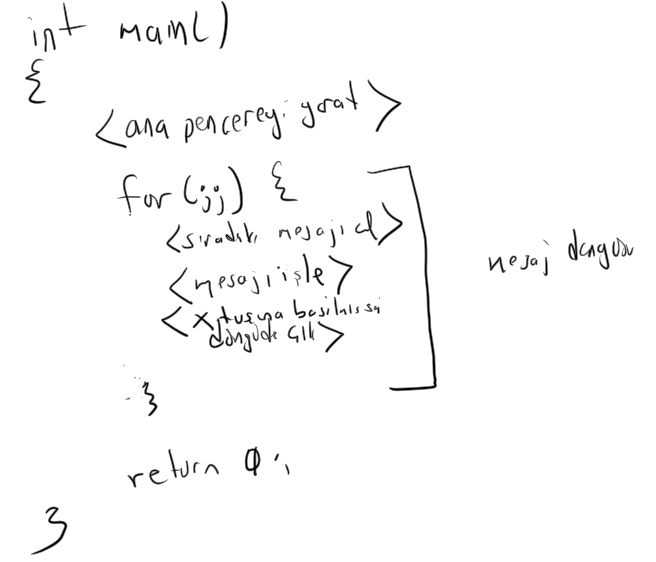
2. İşletim sistemi ya da GUI alt sistem gerçekleşen girdi olaylarını “mesaj (message)” adı altında bir yapı formatına dönüşürmekte ve bunu pencerenin ilişkin olduğu programın (ya da thread’in) mesaj kuyruğuna eklemektedir.

3. Mesajlar ilgili olayı betimleyen ve ona ilişkin bazı bilgileri barındıran yapı (structure) nesleridir. Örneğin Windows’ta mesajlar MSG isimli bir yapıyla temsil edilmişleridir. Bu yapının elemanlarında mesajın ne mesajı olduğu (yani neden gönderildiği) ve olaya ilişkin bazı bilgiler bulunur.

Görüldüğü gibi GUI programlama modelinde girdileri programcı elde etmeye çalışmamaktadır. Girdileri bizzat işletim sisteminin kendisi ya da GUI alt sistemi elde edip programcıya mesaj adı altında iletmektedir.

GUI programlama modelinde işletim sisteminin (ya da GUI alt sistemin) oluşan mesajı ilgili programın (ya da thread’in) mesaj kuyruğuna eklemenin dışında başka bir sorumluluğu yoktur. Mesajların kuyruktan alınarak işlenmesi ilgili programın sorumluluğundadır. Böylece GUI programcısının mesaj kuyruğuna bakarak sıradaki mesajı alması ve ne olmuşsa ona uygun işlemleri yapması gerekir. Bu modelde programcı kodunu şöyle düzenler: Bir döngü içerisinde sıradaki mesajı kuyruktan al, onun neden gönderildiğini belirle, uygun işlemleri yap, kuyrukta mesaj yoksa da blokede bekle”. İşte GUI programlarındaki mesaj kuyruğundan mesajı alıp işleyen döngüye mesaj döngüsü (message loop) denilmektedir.

Bir GUI programının işleyişini tipik akışı aşağıdaki gibi bir kodla temsil edebiliriz:



Bu temsili koddan da görüldüğü gibi tipik bir GUI programında programcı bir döngü içerisinde mesaj kuyruğundan sıradaki mesajı alır ve onu işler. Mesajın işlenmesi ise “ne olmuş ve ben buna karşı ne yapmalıyım?” biçiminde oluşturulmuş olan kodlarla yapılmaktadır.

Pekiyi bir GUI programı nasıl sonlanmaktadır? İşte pencerenin sağındaki (bazı sistemlerde solundaki) X simgesine kullanıcı tıkladığında işletim sistemi ya da GUI alt sistem bunu da bir mesaj olarak o pencerenin ilişkin olduğu prosesin (ya da thread’in) mesaj kuyruğuna bırakır. Programcı da kuyruktan bu mesajı alarak mesaj döngüsünden çıkar ve program sonlanır.

GUI ortamımız ister .NET, ister Java, ister MFC olsun, isterse Qt olsun, işletim sisteminin ya da GUI alt sistemin çalışması hep burada ele açıklandığı gibidir. Yani örneğin biz .NET'te ya da Java'da işlemlerin sanki başka biçimlerde yapıldığını sanabiliriz. Aslında işlemler bu ortamlar tarafından aşağı seviyede yine burada anlatıldığı gibi yapılmaktadır. Bu ortamlar (frameworks) ya da kütüphaneler çeşitli yükleri üzerimizden alarak bize daha rahat bir çalışma modeli sunarlar. Ayrıca şunu da belirtmek istiyoruz: GUI programlama modeli özellikle nesne yönelimli programlama modeline çok uygun düşmektedir. Bu nedenle bu konuda kullanılan kütüphanelerin büyük bölümü sınıflar biçiminde nesne yönelimli diller için oluşturulmuş durumdadır.

Şimdi GUI programlama modelindeki mesaj kavramını biraz daha açalım. Yukarıda da belirttiğimiz gibi bu modelde programcıyı ilgilendiren çeşitli olaylara “mesaj” denilmektedir. Örneğin klavyeden bir tuşa basılması, pencere üzerinde fare ile tıklanması, pencere içerisinde farenin hareket ettirilmesi gibi olaylar hep birer mesaj oluşturmaktadır. İşletim sistemleri ya da GUI alt sistemler mesajları birbirinden ayırmak için onlara birer numara karşılık getirirler. Örneğin Windows’ta mesaj numaraları WM\_XXX biçiminde sembolik sabitlerle kodlanmıştır. Programcılar da konuşurken ya da kod yazarken mesaj numaralarını değil, bu sembolik sabitleri kullanırlar. (Örneğin WM\_LBUTTONDOWN, WM\_MOUSEMOVE, WM\_KEYDOWN gibi) Mesajların numaraları yalnızca gerçekleşen olayın türünü belirtmektedir. Oysa bazı olaylarda gerçekleşen olaya ilişkin bazı bilgiler de söz konusudur. İşte bir mesaja ilişkin o mesaja özgü bazı parametrik bilgiler de işletim sistemi ya da GUI alt sistem tarafından mesajın bir parçası olarak mesajın içerisine kodlanmaktadır. Örneğin Windows’ta biz klavyeden bir tuşa bastığımızda Windows WM\_KEYDOWN isimli mesajı programın mesaj kuyruğuna bırakır. Bu mesajı kuyruktan alan programcı mesaj numarasına bakarak klavyenin bir tuşuna basılmış olduğunu anlar. Fakat hangi tuşa basılmıştır? İşte Windows basılan tuşun bilgisini de ayrıca bu mesajın içerisine kodlamaktadır. Örneğin WM\_LBUTTONDOWN mesajını Windows farenin sol tuşuna tıklandığında kuyruğa bırakır. Ancak ayrıca basım koordinatını da mesaja ekler. Yani bir mesaj oluştuğunda yalnızca o mesajın hangi tür bir olay yüzünden oluştuğu bilgisini değil aynı zamanda o olayla ilgili bazı bilgileri de kuyruktaki mesajın içerisinden alabilmekteyiz.

GUI programlama modelinde bir mesaj oluştuğunda o mesajın bir an evvel işlenmesi ve akışın çok bekletilmemesi gerekir. Aksi takdirde programcı kuyruktaki diğer mesajları işleyemez bu da “program donmuş etkisi” yaratmaktadır. Eğer bir mesaj alındığında uzun süren bir işlem yapılmak isteniyorsa bir thread oluşturulup o işi o thread’e devretmek ve böylece mesaj döngüsünün işlemesini sağlamak gerekir.

GUI programlama modellerinde genel olarak mesaj kavramı pencere kavramıyla ilişkilendirilmiştir. Yani bir pencere yaratılmadıktan sonra bir mesajın oluşma durumu da yoktur. Bu nedenle mesaj döngüsüne girmeden önce programcının en az bir pencere (tipik olarak programın ana penceresi) yaratmış olması gerekir.

Windows gibi bazı sistemlerde thread’lerle ilişkilendirilmiştir. Bu sistemlerde prosesin tek bir mesaj kuyruğu yoktur. Her thread’in ayrı bir mesaj kuyruğu vardır. Bu durumda işletim sistemi ya da GUI alt sistem bir pencereye ilişkin bir işlem gerçekleştiğinde o pencerenin hangi prosesin hangi thread’i tarafından yaratılmış olduğunu belirler ve mesajı o thread’in mesaj kuyruğuna bırakır. Böylece biz bir thread oluşturup o thread’te de bir pencere yaratmışsak artık bizim de o thread’te o pencerenin mesajlarını işlemek için mesaj döngüsü oluşturmamız gerekir. Tabii eğer thread’imizde biz hiçbir pencere oluşturmamışsak böyle bir mesaj döngüsünü oluşturmamıza da gerek yoktur. (Örneğin Microsoft eğer bir thread bir pencere yaratmışsa böyle thread’lere “GUI thread’ler” yaratmamışsa “worker thread’ler” demektedir).

**1.4. Windows Sistemlerinde GUI Programlama Modelinin Temelleri**

Windows sistemlerinde GUI programlama için USER32.DLL içerisindeki API fonksiyonları kullanılmaktadır. (USER32.DLL dosyasının ismindeki 32 sizi yanıltmasın, 64 bit sistemlerde bu DLL aslında isminin aksine zaten 64 bittir.) Bu API fonksiyonları Windows GUI programlama modelinin temellerini bire bir yansıtmaktadır.

**1.4.1. Windows Sistemlerinde İskelet GUI Programı**

Windows sistemlerinde ekrana boş bir pencere çıkartan iskelet GUI programı şöyle yazılabilir:

#include <windows.h>

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam);

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpszCmdParam, int nCmdShow)

{

WNDCLASS wndClass;

HWND hWnd;

MSG message;

if (!hPrevInstance) {

wndClass.style = CS\_HREDRAW | CS\_VREDRAW;

wndClass.cbClsExtra = 0;

wndClass.cbWndExtra = 0;

wndClass.hInstance = hInstance;

wndClass.hIcon = LoadIcon(NULL, IDI\_QUESTION);

wndClass.hbrBackground = GetStockObject(WHITE\_BRUSH);

wndClass.hCursor = LoadCursor(NULL, IDC\_ARROW);

wndClass.lpszMenuName = NULL;

wndClass.lpszClassName = "Generic";

wndClass.lpfnWndProc = (WNDPROC)WndProc;

if (!RegisterClass(&wndClass))

return -1;

}

hWnd = CreateWindow("Generic", "Sample Windows", WS\_OVERLAPPEDWINDOW, CW\_USEDEFAULT, 0,

CW\_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);

if (!hWnd)

return -1;

ShowWindow(hWnd, SW\_RESTORE);

UpdateWindow(hWnd);

while (GetMessage(&message, 0, 0, 0)) {

TranslateMessage(&message);

DispatchMessage(&message);

}

return message.wParam;

}

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)

{

switch (message) {

case WM\_DESTROY:

PostQuitMessage(0);

break;

default:

return DefWindowProc(hWnd, message, wParam, lParam);

}

return 0;

}

Windows API programlamda kullanılan çeşitli typedef türleri ve sembolik sabitler vardır. Tüm bu typedef’ler ve sembolik sabitler ve API fonksiyonlarının prototipleri <windows.h> dosyasının içerisindedir. API programlamada ağırklı yazım biçimi olarak macar notasyonu (Hungarian Notation) kullanılmaktadır. Macar Notasyonunda değişkenler onların türlerini belirten küçük harf öneklerle başlatılır, sonra Pascal tarzı harflendirme ile devam edilir. Örneğin:

long lNumberOfSectors;

HINSTANCE hInstance;

DWORD dwFlags;

char szPath[MAX\_PATH];

Türler için önemli kullanılan önekler şunlardır:

|  |  |
| --- | --- |
| p ya da lp | Gösterici (lp long pointer'dan gelme. Eskiye uyum için hala kullanılıyor.) |
| l | long |
| w | WORD |
| dw | DWORD |
| h | HANDLE |
| sz | char \* (fakat yazı gösterir) |
| b | BOOL |
| f | float |
| d | double |

API programlamada kullanılan typedef tür isimleri hep büyük harflerle oluşturulmuştur. En çok kullanılanları şunlardır:

|  |  |
| --- | --- |
| BYTE | Bir byte'lık işaretsiz tamsayı türü (unsigned char) |
| WORD | İki byte'lık işaretsiz tamsayı türü (unsigned short int) |
| DWORD | Dört byte'lık işaretsiz tamsayı türü (unsigned long int ya da unsigned int) |
| HANDLE | Handle türü (void \*) |
| HXXXX | Handle türü (void \*) |
| PXXX, LPXXX | XXX türünden adres türü (örneğin LPINT, LPVOID, PVOID, LPDWORD) |
| PCXXX, LPCXXX | XXX türünden gösterdiği yer const olan adres (Örneğin LPCVOID demek const void \* demektir) |
| LPSTR | Yazıyı gösteren adres (char \*) |
| LPTSTR | Yazıyı gösteren UNICODE destekli adres (char \* ya da wchar\_t \*) |
| BOOL | int türünü belirtir. Fakat anlam olarak başarı ve başarısızlık düşünülmelidir. Geri dönüş değeri BOOL olan API fonksiyonları başarı durumunda sıfır dışı değere, başarızılık durumunda sıfır değerine geri dönerler. |

Fonksiyon prototiplerinde parametre değişkenlerinin önündeki \_\_in, \_\_out ve \_\_in\_out makroları okunabilirliği artırmak için düşünülmüştür. Bu makrolar aşağıdaki gibi define edilmiştir:

#define \_\_in

#define \_\_out

#define \_\_in\_out

Görüldüğü gibi aslında bu makrolar önişlemci tarafından silinmektedir. \_\_in makrosu fonksiyonun parametre değişkenindeki bilgiyi kullanacağı fakat ona bir değer yerleştirmeyeceği anlamına gelir. \_\_out tam tersine fonksiyonun parametre değişkenindeki değeri değiştireceği anlamına gelmektedir. \_\_in\_out ise fonksiyonun hem parametre değişkenindeki değeri kullanacağı hem de ona yeni bir değer yerleştireceği anlamına gelir. Tabii bu makroların gerekliliği tartışılabilir. (Bilindiği gibi zaten gösterici olmayan parametre değişkenleri \_\_in olmak zorundadır. Gösterici parametre değişkenlerinde \_\_in ya da \_out durumu göstericinin const olup olmamasıyla da zaten anlaşılmaktadır. O halde bu makrolar yalnızca \_\_in\_out durumunun anlaşılmasına katkıda bulunmaktadır.)

Yukarıdaki typedef isimlerinin dışında <Windows.h> dosyası üzerinde pek çok sembolik sabit ve makro da bulunmaktadır. Ancak bunların hepsini burada ele almamız imkansız. Fakat sembolik sabitler genel olarak onların hangi amaçla bulundurulduğuna ilişkin öneklere sahip olduğunu belirtelim (örneğin WS\_CHILD, WM\_COMMAND gibi). İleride kullanacağımız iki makroyu da burada tanıtmayı uygun görüyoruz. istiyoruz: LOWORD makrosu 4 byte’lık bir değerin bize düşük anlamlı 2 byte’ını, HIWORD makrosu da yüksek anlamlı 2 byte’ını vermektedir.

**1.4.1.1. Windows Sistemlerinde İskelet GUI Programının Açıklaması**

Windows GUI uygulamalarında programın başlangıç noktası (entry point) main isimli fonksiyon değil WinMain isimli fonksiyondur. WinMain fonksiyonunun dört parametresi vardır. Bu parametrelerin anlamları şöyledir:

HINSTANCE hInstance: Bu parametre programın (yani PE formatının) bellekteki yüklenme adresini belirtir. (16 bit Windows sistemlerinde bu adres prosesin “modül veritabanı (module database)” adresini belirtiyordu.)

HINSTANCE hPrevInstance: 16 bit Windows sistemlerinde program birden fazla kez çalıştırılmışsa bu parametre önceki çalıştırmaya ilişkin modül veritabanı adresini belirtiyordu. Ancak bu parametre 32 bit sistemlerde ve 64 bit sistemlerde bu değer her zaman NULL değerinde olmaktadır.

LPSTR lpszCmdParam: Bu parametre programın komut satırı argümanlarını belirtmektedir. Windows GUI uygulamalarında program ismi dahil olmak üzere tüm komut satırı argümanları tek bir yazı biçiminde WinMain fonksiyonuna aktarılmaktadır.

int nCmdShow: Bu parametre programın ana penceresinin nasıl görüntüleneceği konusunda bir tavsiye niteliğindedir. Bu değer PE formatından alınarak WinMain fonksiyonuna aktarılmaktadır.

İskelet GUI programında ilk olarak bir pencere sınıfının register ettirilmiştir:

if (!hPrevInstance) {

wndClass.style = CS\_HREDRAW | CS\_VREDRAW;

wndClass.cbClsExtra = 0;

wndClass.cbWndExtra = 0;

wndClass.hInstance = hInstance;

wndClass.hIcon = LoadIcon(NULL, IDI\_QUESTION);

wndClass.hbrBackground = GetStockObject(WHITE\_BRUSH);

wndClass.hCursor = LoadCursor(NULL, IDC\_ARROW);

wndClass.lpszMenuName = NULL;

wndClass.lpszClassName = "Generic";

wndClass.lpfnWndProc = (WNDPROC)WndProc;

if (!RegisterClass(&wndClass))

return -1;

}

16 bit Windows sistemlerinde bu register ettirme işlemi yalnızca programın ilk kopyası çalıştırıldığında yapılmak zorundaydı. Halbuki 32 bit ve 64 bit sistemlerde her zaman yapılmak zorundadır. (Zaten hPrevInstance değerinin 32 bit ve 64 bit Windows sistemlerine her zaman NULL olduğunu anımsayınız). Dolayısıyla buradaki if kontrolüne 32 bit ve 64 bit Windows programlarında artık hiç gerek yoktur. Ancak geriye doğru uyum için pek çok programcı bu kontrolü hala bulundurmaktadır.

Pencere sınıfının register ettirilmesi ne anlama gelir? Bir pencerenin pek çok özelliği vardır. Bu özelliklerden bazıları pencere sınıfı denilen bir yapı ile temsil edilir. Bir isim altında oluşturulur. Sonra pencere yaratılırken bu isim kullanılır. Böylece pencerenin bazı bilgileri önceden yapılmış bu belirlemelerden alınır. Pencere sınıfı (buradaki sınıfın C++’taki sınıf ile bir ilgisi yoktur) WNDCLASS isimli bir yapıyla temsil edilmiştir:

typedef struct {

UINT style;

WNDPROC lpfnWndProc;

int cbClsExtra;

int cbWndExtra;

HINSTANCE hInstance;

HICON hIcon;

HCURSOR hCursor;

HBRUSH hbrBackground;

LPCTSTR lpszMenuName;

LPCTSTR lpszClassName;

} WNDCLASS, \*PWNDCLASS;

Bu yapıda pencereler için şu belirlemeler yapılmaktadır:

- Pencere fonksiyonunun hangi fonksiyon olacağı

- Fare pencere üzerindeyken görüntülenecek fare oku (cursor)

- Pencerenin simgesi (icon)

- Pencerenin zemin rengi

- Pencere stillerinin nasıl olacağı

İskelet programda pencere sınıfı RegisterClass API fonksiyonuyla sisteme register ettirilmiştir. RegisterClass API fonksiyonunun RegisterClassEx isimli genişletilmiş bir biçimi de vardır. Bu genişletilmiş biçim WNDCLASS yerine WNDCLASSEX yapısını parametre olarak almaktadır.

İskelet programda pencere sınıfı register ettirildikten sonra sıra ana pencerenin yaratılmasına gelmiştir. Windows'ta her türlü pencere (üst pencereler, ana pencereler ya da dialog pencereleri) CreateWindow API fonksiyonuyla yaratılmaktadır. CreateWindow fonksiyonunun daha sonradan CreateWindowEx isimli genişletilmiş bir biçimi de oluşturulmuştur. CreateWindow fonksiyonunun prototipi şöyledir:

HWND CreateWindow(LPCTSTR lpClassName,

LPCTSTR lpWindowName,

DWORD dwStyle,

int x,

int y,

int nWidth,

int nHeight,

HWND hWndParent,

HMENU hMenu,

HINSTANCE hInstance,

LPVOID lpParam

);

Fonksiyonun birinci parametresi pencerenin hangi pencere sınıfından yaratılacağını belirtir. (Anımsanacağı gibi pencere sınıfı bir isim altında register ettiriliyordu. İşte birinci parametre bu ismi alıyor.) İkinci parametre pencere yazısını belirtmektedir. Windows'ta her pencerenin bir pencere yazısı vardır. (Örneğin ana pencerelerde bu yazı pencere başlığında görüntülenecek olan yazıdır, düğmelerin (buttons) pencere yazısı düğmenin üzerindeki yazıdır. Seçenek kutularının yazısı küçük kutucuğun yanındaki yazıdır vs.) Üçüncü parametre pencere stillerini belirtir. Pencere stilleri <Windows.h> dosyası içerisinde WS\_XXX biçimindeki sembolik sabitlerle bit düzeyinde bayraklarla define edilmiştir. Örneğin eğer alt pencere yaratılacaksa burada WS\_CHILD bayrağının kullanılması gerekir. Fonksiyonun sonraki dört parametresi pencerenin ilk açıldığındaki konumunu ve büyüklüğünü alır. Fonksiyonun hWndParent parametresi ise eğer pencere ana pencereyse NULL olarak, alt pencereseyse onun üst penceresinin handle değeri olarak girilmelidir. hMenu parametresi pencerenin menüsü varsa o menü handle’ını belirtir. Penceresinin menüsü yoksa bu parametre de NULL biçiminde girilmelidir. hInstance parametresi WinMain’e geçirilen hInstance değerini alır. Son parametre isteğe bağlı bir değerdir. Bazı pencere sınıfları buraya bir yapı adresinin geçirilmesini isterler. Fonksiyonun ayrıntılı açıklaması için MSDN dokümanlarına başvurabilirsiniz. İskelet programda programın ana penceresi şöyle yaratılmıştır:

hWnd = CreateWindow("Generic", "Sample Windows", WS\_OVERLAPPEDWINDOW, CW\_USEDEFAULT, 0,

CW\_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);

Windows’ta her pencerenin sistem genelinde tek olan bir handle değeri vardır. CreateWindow API fonksiyonu başarı durumunda bu handle değerine, başarısızlık durumunda ise NULL değerine geri döner.

**Anahtar Notlar:** Sistemdeki tüm prosesleri, pencereleri ve bu pencerelere gelen mesajları görüntüleyen Spy++ isimli bir araç vardır. Spy++ Microsoft tarafından geliştirilmiştir ve Visual Studio paketinin içerisindedir.

CreateWindow fonksiyonu ile ana pencere yaratıldıktan sonra o henüz görünür değildir. Onu görünür hale getirmek isiçin ShowWindow API fonksiyonu kullanılmalıdır. Ayrıca iskelet programdaki UpdateWindow çağrısının da mutlak gerekli olmadığını belirtelim. UpdateWindow fonksiyonu pencere fonksiyonunun çizim için çağrılmasına yol açmaktadır.

İskelet programda ana pencere görünür hale getirildikten sonra artık mesaj döngüsüne girilmiştir. Mesaj döngüsünün aşağıdaki gibi oluşturulmuştur:

while (GetMessage(&message, 0, 0, 0)) {

TranslateMessage(&message);

DispatchMessage(&message);

}

Mesaj döngüsünden sıradaki mesajı alan API fonksiyonu GetMessage fonksiyonudur. GetMessage mesajı kuyruktan alarak MSG isimli bir yapıya yerleştirmektedir. Windows’ta mesajlar en aşağı seviyeli olarak MSG isimli yapıyla temsil edilmektedir. GetMessage fonksiyonunun diğer parametreleri hangi aralıktaki mesajların alınacağına yöneliktir:

BOOL GetMessage(LPMSG lpMsg,

HWND hWnd,

UINT wMsgFilterMin,

UINT wMsgFilterMax

);

MSG yapısı da şöyledir:

typedef struct {

HWND hwnd;

UINT message;

WPARAM wParam;

LPARAM lParam;

DWORD time;

POINT pt;

} MSG, \*PMSG;

Yapının hWnd elemanı mesajın hangi pencereye gönderildiğini belirtir. Daha önceden de belirttiğimiz gibi Windows'ta mesajlar sistem tarafından belli bir pencere için gönderilmektedir. Windows'ta ister ana pencere olsun isterse herhangi bir alt pencere olsun bir thread’in yarattığı tüm pencerelerin mesajları aynı mesaj kuyruğuna (o thread’in mesaj kuyruğuna yerleştirilirler.

İskelet programda kuyruktan mesaj alındıktan sonra mesaj önce "translate" edilmiştir, sonra da "disptach" edilmiştir. Aslında mesajı "translate" etmek (yani TranslateMessage fonksiyonunu çağırmak) mutlak anlamda zorunlu değildir. Ancak mesaj DispatchMessage fonksiyonu çağrılarak "dispatchg edilmelidir. TranslateMessage WM\_KEYDOWN mesajları için WM\_CHAR mesajını oluşturarak bazı encoding dönüştürmelerini yapmaktadır. DispatchMessage fonksiyonunun kuyruktan alınan mesaj argüman yapılarak çağrıldığına dikkat ediniz. DispatchMessage ileride de göreceğimiz gibi pencere sınıfına ilişkin "pencere fonksiyonu" denilen bir fonksiyonun çağrılmasını sağlamaktadır.

Pekiyi mesaj döngüsünden nasıl çıkılmaktadır. İskelet programda ancak GetMessage fonksiyonu 0 ile (FALSE ile) geri dönerse mesaj döngüsünden çıkılabilmektedir. İşte GetMessage fonksiyonu kuyruktan WM\_QUIT isimli mesajı aldığında 0 ile geri döner. O halde şöyle söyleyebiliriz: İskelet programda WM\_QUIT mesajı kuyruğa bırakıldığında mesaj döngüsünden çıkılmaktadır. WM\_QUIT mesajının oluşturulmasdı ve işlevi ileride ele alınacaktır. Önce pencere fonksiyonu üzerinde durmam istiyoruz.

**1.4.2. Pencere Sınıflarına İlişkin Pencere Fonksiyonları**

Bir pencereye bir mesaj gönderildiğinde çağrılması istenilen fonksiyona pencere fonksiyonu denilmektedir. Pencere fonksiyonun hangi fonksiyon olacağı pencere sınıfı register ettirilirken (WNDCLASS yapısında) belirlenir. Sonra pencere o sınıf kullanılarak yaratıldığında artık çağrılacak fonksiyon da belirlenmiş olmaktadır. Tabii kuyruktan mesaj alındığında pencere fonksiyonu otomatik çağrılmaz. Yukarıda da sözünü ettiğimiz gibi pencere fonksiyonun çağrılmasına DisptachMessage fonksiyonu yol açmaktadır. (Ancak DispatchMessage pencere fonksiyonun pencere fonksiyonunu çağırmanının dışında bazı diğer işlemleri de yaptığını belirtelim).

Pencere fonksiyonu herhangi bir fonksiyon olamaz. Pencere fonksiyonunun parametrik yapısının ve geri dönüş değerinin nasıl olması gerektiği WNDPROC isimli typedef türüyle belirlenmiştir:

typedef LRESULT (CALLBACK \*WNDPROC)(HWND, UINT, WPARAM, LPARAM);

LRESULT long türünü belirtir. CALLBACK makrosu da \_\_stdcall olarak define edilmiştir. (Yani pencere fonksiyonlarının çağırma biçimi (calling convention) \_\_stdcall olmak zorundadır.)

Pencere fonksiyonunun birinci parametresi mesajın gönderildiği (yani ilişkin olduğu) pencerenin handle değerini belirtir. İkinci parametre gönderilen mesajın numarasıdır. Mesaj numaralarının <windows.h> içersinde WM\_XXX biçiminde sembolik sabitlerle define edildiğini daha önce belirtmiştik. Üçüncü ve dördüncü parametreler mesaja ilişkin ekstra bilgileri belirtmektedir. Üçüncü parametre WPARAM türünden dörüdüncü parametre LPARAM türdendir. WPARAM 2 byte'lık işaretsiz tamsayı türü olarak, LPARAM ise 4 byte'lık işaretli tamsayı türü olarak typedef edilmişleridir. Pekiyi DispatchMessage pencere fonksiyonunu çağırırken bu parametreler için argümanları nasıl oluşturmaktadır? Aslında pencere fonksiyonuna aktarılan bu bilgilerin hepsi zaten kuyruktan alınan mesajın içerisinde (yani MSG yapısında) bulunmaktadır. DispatchMessage yalnızca onları mesajın içerisinden alarak pencere fonksiyonuna arüman yapmaktadır.

Bir pencereye mesaj geldiğinde o pencereye ilişkin pencere sınıfında belirtilen pencere fonksiyonunun çağrıldığını gördük. Pekiyi pencere fonksiyonu içerisinde mesajları nasıl işleyeceğiz? İşte bunun için en uygun yol mesaj numarasını switch içerisine almaktır. İskelet programdaki pencere fonksiyonunda da böyle yapıldığını görüyorsunuz:

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)

{

switch (message) {

case WM\_DESTROY:

PostQuitMessage(0);

break;

default:

return DefWindowProc(hWnd, message, wParam, lParam);

}

return 0;

}

Windows'ta bazı mesajlarda bazı kritik işlemlerin yapılması gerekebilmektedir. Bu mesajların işlenmesi programcı için çok zahmetli olacağından DefWindowProc isimli bir fonksiyon bulundurulmuştur. Böylece programcı işlemediği mesajları DefWindowProc fonksiyonuna verir, mesajı onun işlemesini sağlar. Bazı mesajlar için DefWindowProc hiçbir şey yapmamaktadır. Ancak bazıları için yukarıda belirtildiği gibi bazı önemli işlemleri yapar. Pencere fonksiyonundan normal olarak sıfır ile geri dönülmelidir. Ancak bazı durumlarda bazı özel değerlerle geri dönülmesi gerekebilmektedir.

**1.4.3. Windows GUI Programlarının Sonlandırılması**

Windows’ta bir GUI programı tipik olarak şu adımlardan geçilerek sonlandırılır:

1) Kullanıcı ana pencerenin X simgesine tıklar ya da Alt + F4 tuşlarına basar. Bu durumda Windows kuyruğa o pencere için WM\_CLOSE mesajını bırakır.

2) WM\_CLOSE mesajını alan programcı tipik olarak (ancak zorunlu değil) DestroyWindow fonksiyonunu çağırır. (Bir pencere nasıl CreateWindow fonksiyonuyla yaratılıyorsa DestroyWindow fonksiyonuyla da yok edilmektedir.) Eğer programcı WM\_CLOSE mesajını işlemezse DefWindowProc bu mesaj için zaten DestroyWindow fonksiyonunu çağırmaktadır. (Örneğin bu durumda örneğin X tuşuna basıldığında DestroyWindow fonksiyonun çağrılmasını engellersek pencere kapatılmaz.)

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)

{

switch (message) {

case WM\_CLOSE:

MessageBox(NULL, "You cannot close this window!", "Warning", MB\_OK);

break;

case WM\_DESTROY:

PostQuitMessage(0);

break;

default:

return DefWindowProc(hWnd, message, wParam, lParam);

}

return 0;

}

3) DestroyWindow fonksiyonu çağrıldığında bu fonksiyon pencereyi yok etmeden hemen önce pencereye WM\_DESTROY mesajını yollar. Programcılar pencere yaratılırken eğer bazı tahsisatlar yapmışlarsa onları tipik olarak WM\_DESTROY mesajında serbest bırakırlar.

4) WM\_DESTROY mesajı için DefWindowProc birşey yapmaz. Tipik olarak eğer ana pencere yok edilmişse programcılar bu mesajda PostQuitMesssage API fonksiyonunu çağırırlar. PostQuitMessage fonksiyonu kuyruğa WM\_QUIT mesajını bırakakmaktadır.

5) WM\_QUIT mesajını alan GetMessage API fonksiyonu 0 ile geri döner ve mesaj döngüsünden çıkılır. Böylece WinMain sonlanır program da bitmiş olur.

**1.4.4. Pencere Mesajlarının İşlenmesi**

Daha önceden de belirttiğimiz gibi mesajlar tipik olarak pencere fonksiyonun içerisinde mesaj numarasının switch içerisine alınmasıyla işlenmektedir. Tabii okunabilirlik gereği switch deyiminin case bölümünde uzun işlemler yapılacaksa o işlemlerin bir fonksiyona havale edilmesi daha uygun olur. Böylece kod biraz daha okunabilir hale getirilebilir. Mesajların bu biçimde işlenmesi tipik olarak prosedürel programama tekniğini akla getirmektedir. Örneğin farenin sol tuşuna basıldığında bir MessageBox çıkartmak isteyelim:

void LButtonDownHandler(HWND hWnd, WPARAM wParam, LPARAM lParam)

{

char buf[100];

sprintf(buf, "X = %u, Y = %u", LOWORD(lParam), HIWORD(lParam));

MessageBox(hWnd, buf, "Info", MB\_OK);

}

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)

{

switch (message) {

case WM\_LBUTTONDOWN:

LButtonDownHandler(hWnd, wParam, lParam);

break;

case WM\_DESTROY:

PostQuitMessage(0);

break;

default:

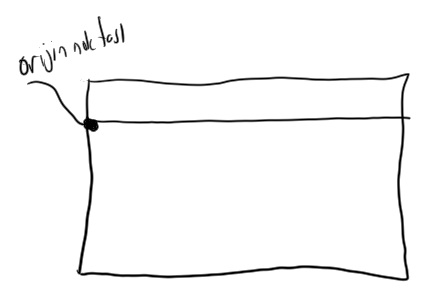
return DefWindowProc(hWnd, message, wParam, lParam);

}

return 0;

}

WM\_LBUTTONDOWN mesajında lParam parametresinin düşük anlamlı WORD kısmı basım yerinin X koordinatını, yüksek anlamlı WORD kısmı basım yerinin y koordinatını belirtir. Buradaki değerler çalışma alanı orijinlidir. Yani orijin noktası pencere başlığının altındaki sınır çizgilerinin içerisindeki bölgenin sol üst köşesidir:



**1.4.5. Alt Pencerelerin Yaratılması**

Windows'ta tüm pencereler CreateWindow fonksiyonuyla yaratılırlar. Alt pencerelerin yaratımı sırasında CreateWindow fonksiyonunun hWndParent parametresi üst pencerenin handle değeri olarak girilir. Ayrıca fonksiyonun ikinci parametresinde WS\_CHILD belirlemesinin yapılması gerekir. Alt pencerelerin için ayrı bir pencere sınıfı kullanılarak yaratılması daha uygundur. Bu durumda o alr pencereye gelen mesajlar ayrı bir pencere fonksiyonu tarafından işlenecektir. Alt pencerelerin yaratımı herhangi bir yerde yapılabilmektedir. Ancak en uygun yer üst pencerenin WM\_CREATE mesajıdır. Bir pencere yaratılır yaratılmaz pencere henüz görünür halde değilken Windows onun üst penceresine WM\_CREATE isimli bir mesajı gönderilmektedir.

Örnek bir alt pencere yaratımı şöyle yapılabilmektedir:

#include <windows.h>

LRESULT CALLBACK WndProcParent(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam);

LRESULT CALLBACK WndProcChild(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam);

HWND g\_hWndParent;

HWND g\_hWndChild;

HINSTANCE g\_hInstance;

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpszCmdParam, int nCmdShow)

{

WNDCLASS wndClassParent, wndClassChild;

MSG message;

if (!hPrevInstance) {

wndClassParent.style = CS\_HREDRAW | CS\_VREDRAW;

wndClassParent.cbClsExtra = 0;

wndClassParent.cbWndExtra = 0;

wndClassParent.hInstance = hInstance;

wndClassParent.hIcon = LoadIcon(NULL, IDI\_QUESTION);

wndClassParent.hbrBackground = GetStockObject(WHITE\_BRUSH);

wndClassParent.hCursor = LoadCursor(NULL, IDC\_ARROW);

wndClassParent.lpszMenuName = NULL;

wndClassParent.lpszClassName = "GenericParent";

wndClassParent.lpfnWndProc = (WNDPROC)WndProcParent;

if (!RegisterClass(&wndClassParent))

return -1;

wndClassChild.style = CS\_HREDRAW | CS\_VREDRAW;

wndClassChild.cbClsExtra = 0;

wndClassChild.cbWndExtra = 0;

wndClassChild.hInstance = hInstance;

wndClassChild.hIcon = LoadIcon(NULL, IDI\_HAND);

wndClassChild.hbrBackground = CreateSolidBrush(RGB(255, 0, 0));

wndClassChild.hCursor = LoadCursor(NULL, IDC\_CROSS);

wndClassChild.lpszMenuName = NULL;

wndClassChild.lpszClassName = "GenericChild";

wndClassChild.lpfnWndProc = (WNDPROC)WndProcChild;

if (!RegisterClass(&wndClassChild))

return -1;

}

g\_hInstance = hInstance;

g\_hWndParent = CreateWindow(GenericParent", "Sample Windows", WS\_OVERLAPPEDWINDOW, CW\_USEDEFAULT, 0,

CW\_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);

if (!g\_hWndParent)

return -1;

ShowWindow(g\_hWndParent, SW\_RESTORE);

UpdateWindow(g\_hWndParent);

while (GetMessage(&message, 0, 0, 0)) {

TranslateMessage(&message);

DispatchMessage(&message);

}

return message.wParam;

}

LRESULT CALLBACK WndProcParent(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)

{

switch (message) {

case WM\_CREATE:

g\_hWndChild = CreateWindow("GenericChild", "", WS\_CHILD | WS\_VISIBLE | WS\_BORDER, 10, 10,

100, 100, hWnd, NULL, g\_hInstance, NULL);

if (!g\_hWndChild) {

MessageBox(hWnd, "Cannot create child!", "Error", MB\_OK);

return -1;

}

break;

case WM\_LBUTTONDOWN:

MessageBox(hWnd, "Parent WM\_LBUTTONDOWN", "Message", MB\_OK);

break;

case WM\_DESTROY:

PostQuitMessage(0);

break;

default:

return DefWindowProc(hWnd, message, wParam, lParam);

}

return 0;

}

LRESULT CALLBACK WndProcChild(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)

{

switch (message) {

case WM\_LBUTTONDOWN:

MessageBox(hWnd, "Child WM\_LBUTTONDOWN", "Message", MB\_OK);

break;

default:

return DefWindowProc(hWnd, message, wParam, lParam);

}

return 0;

}

Alt pencerenin ayrı pencere sınıfı ile yaratıldığına ve dolayısıyla alt pencerenin ayrı pencere fonksiyonuna sahip olduğuna dikkat ediniz. Alt pencereyi yaratırken CreateWindow fonksiyonunun ikinci parametresi WS\_CHILD|WS\_VISIBLE|WS\_BORDER biçiminde girildiğine dikkat ediniz. WS\_VISIBLE yaratılan alt pencerenin yaratılır yaratılmaz görünür hale getirileceğini belirtmektedir. (Eğer yaratım sırasında WS\_CHILD bayrağı kullanılmazsa alt pencerenin ShowWindow ile görünür hale getirilmesi gerekir). WS\_BORDER ise yaratılan alt pencerenin sınır çizgilerine sahip olacağını belirtmektedir.

**1.4.6. Pencerelere Mesaj Gönderilmesi**

Yalnızca işletim sisteminin ya daa GUI alt sistemin kendisi değil, programcılar da isterlerse pencerelere mesajlar gönderebilirler. Mesaj göndermenin iki yolu vardır: SendMessage API fonksiyonu ile ya da PostMessage API fonksiyonu ile. (Windows dokümanlarında açıklamaklar yapılırken “sends” sözcüğü görüldüğünde mesajın SendMessage fonksiyonu ile “posts” sözcüğü görüldüğünde ise PostMessage ile gönderildiği anlaşılmalıdır.) SendMessage ve PostMesssage fonksiyonlarının çok benzerdir:

LRESULT SendMessage(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam);

BOOL PostMessage(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam);

SendMessage doğrudan pencere fonksiyonunu çağırarak mesajı işler. PostMessage ise mesajı mesaj kuyruğuna bırakır. SendMessage doğrudan pencere fonksiyonunun çağırdığı için akış SendMessage fonksiyonundan geri döndüğünde pencere fonksiyonu çalıştırımış ve mesaj işlenmiş olur. Örneğin:

SendMessage(hWnd, WM\_USER, 0, 0);

...

Burada SendMessage fonksiyonu geri döndüğünde mesaj işlenmiş durumdadır. Halbuki PostMessage fonksiyonu mesajı ilgili pencereyi yaratan thread’in mesaj kuyruğuna bırakmaktadır:

PostMessage(hWnd, WM\_USER, 0, 0);

Burada akış PostMessage fonksiyonundan çıktığında henüz mesaj işlenmiş değildir. Messaj hangi pencereye gönderilmişse ilgili mesaj döngüsü yoluyla mesaj kuyruktan GetMessage fonksiyonuyla alınıp “dispatch” edilerek işlenecektir. Dolayısıyla SendMessage mesajın “senkron” olarak, PostMessage ise “asenkron” olarak işlenmesine yol açar. (Örneğin CreateWindow fonksiyonu WM\_CREATE mesajını, DestroyWindow ise WM\_DESTROY mesajlarını SendMessage yoluyla göndermektedir.)

**1.4.7. Standart Alt Pencerelerin Kullanılması**

Windows’ta görsel arayüzü oluşturmak için sistem tarafından düğme gibi, edit alanı gibi, listeleme kutuları gibi çeşitli alt pencere sınıfları oluşturulup register ettirilmiştir. Bu pencere sınıflarına ilişkin pencere fonksiyonları Windows’un USER32.DLL dosyasının içerisindedir. Böylece programcı bu görsel elemanlara verilen sınıf isimlerini kullanarak bu standart alt pencereleri CreateWindow fonksiyonuyla yaratabilir. Aslında düğmeler, edit alanları, listeleme kutuları gibi görsel öğeler boş bir pencere üzerinde yapılan çizim işlemleriyle ve bazı mesajların işlenmesiyle gerçekleştirilmişlerdir. Örneğin “button” sınıfı normal düğmeleri (push button), radyo düğmelerini (radio buttons) ve seçenek kutularını (check boxes) yaratmak için, “edit” sınıfı edit alanı (edit box) yaratmak için, “static” sınıfı yalnızca yazı bulunan yazı penceresi (label) yaratmak için kullanılabilir. Örnek bir standart alt pencere uygulaması şöyle verilebilir:

#include <windows.h>

#define ID\_BUTTONOK 100

#define ID\_BUTTONCANCEL 101

#define ID\_EDITNAME 102

#define ID\_EDITNO 103

LRESULT CALLBACK WndProcParent(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam);

LRESULT CALLBACK WndProcChild(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam);

HWND g\_hWndParent;

HWND g\_hButtonOk;

HWND g\_hButtonCancel;

HWND g\_hEditBoxName;

HWND g\_hEditBoxNo;

HWND g\_hStaticName;

HWND g\_hStaticNo;

HINSTANCE g\_hInstance;

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpszCmdParam, int nCmdShow)

{

WNDCLASS wndClassParent, wndClassChild;

MSG message;

if (!hPrevInstance) {

wndClassParent.style = CS\_HREDRAW | CS\_VREDRAW;

wndClassParent.cbClsExtra = 0;

wndClassParent.cbWndExtra = 0;

wndClassParent.hInstance = hInstance;

wndClassParent.hIcon = LoadIcon(NULL, IDI\_QUESTION);

wndClassParent.hbrBackground = GetStockObject(WHITE\_BRUSH);

wndClassParent.hCursor = LoadCursor(NULL, IDC\_ARROW);

wndClassParent.lpszMenuName = NULL;

wndClassParent.lpszClassName = "GenericParent";

wndClassParent.lpfnWndProc = (WNDPROC)WndProcParent;

if (!RegisterClass(&wndClassParent))

return -1;

}

g\_hInstance = hInstance;

g\_hWndParent = CreateWindow("GenericParent", "Sample Windows", WS\_OVERLAPPEDWINDOW, CW\_USEDEFAULT, 0,

450, 200, NULL, NULL, hInstance, NULL);

if (!g\_hWndParent)

return -1;

ShowWindow(g\_hWndParent, SW\_RESTORE);

UpdateWindow(g\_hWndParent);

while (GetMessage(&message, 0, 0, 0)) {

TranslateMessage(&message);

DispatchMessage(&message);

}

return message.wParam;

}

LRESULT CALLBACK WndProcParent(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)

{

switch (message) {

case WM\_CREATE:

g\_hButtonOk = CreateWindow("button", "Ok", WS\_CHILD | WS\_BORDER | WS\_VISIBLE | BS\_PUSHBUTTON , 260, 120,

70, 25, hWnd, (LPVOID)ID\_BUTTONOK, g\_hInstance, NULL);

g\_hButtonCancel = CreateWindow("button", "Cancel", WS\_CHILD | WS\_BORDER | WS\_VISIBLE | BS\_PUSHBUTTON, 340, 120,

70, 25, hWnd, NULL, g\_hInstance, NULL);

g\_hStaticName = CreateWindow("static", "Adı Soyadı", WS\_CHILD | WS\_VISIBLE, 10, 10,

100, 20, hWnd, (LPVOID)ID\_BUTTONCANCEL, g\_hInstance, NULL);

g\_hStaticNo = CreateWindow("static", "No", WS\_CHILD | WS\_VISIBLE, 10, 60,

100, 20, hWnd, NULL, g\_hInstance, NULL);

g\_hEditBoxName = CreateWindow("edit", "", WS\_CHILD | WS\_BORDER | WS\_VISIBLE, 10, 30,

200, 20, hWnd, (LPVOID)ID\_EDITNAME, g\_hInstance, NULL);

g\_hEditBoxNo = CreateWindow("edit", "", WS\_CHILD | WS\_BORDER | WS\_VISIBLE, 10, 80,

200, 20, hWnd, (LPVOID)ID\_EDITNO, g\_hInstance, NULL);

break;

case WM\_LBUTTONDOWN:

MessageBox(hWnd, "Parent WM\_LBUTTONDOWN", "Message", MB\_OK);

break;

case WM\_DESTROY:

PostQuitMessage(0);

break;

default:

return DefWindowProc(hWnd, message, wParam, lParam);

}

return 0;

}

**1.4.8. Alt Pencere Mesajlarının İşlenmesi**

Windows’ta bir ister ana pencere olsun ister alt pencere olsun bir thread’in yarattığı tüm pencerelere gönderilen (“post” edilen) mesajlar aynı mesaj kuyruğuna bırakılırlar ve aynı mesaj döngüsü tarafından ele alınırlar. Ayrıca belli bir girdi olayı için mesajlar o girdi olayına ilişkin olan tek bir pencereye gönderilmektedir. Örneğin bir alt pencere üzerinde fareye tıkladığımızda fare mesajları yalnızca bu alt pencereye yollanacaktır.

Pencere fonksiyonlarının pencere sınıflarında belirlendiğini anımsayınız. Bu durumda eğer biz yarattığımız bir alt pencereye ilişkin alt pencere sınıfını kendimiz oluşturmuşsak -onun pencere fonksiyonunu da kendimiz yazmış olacağımızdan- o alt pencereye gönderilen mesajları doğrudan işleyebiliriz. Pekiyi biz eğer Windows’un standart bir alt pencere sınıfını (ya da başkalarının oluşturup register ettirdiği) kullandığımızda o pencereye gönderilen mesajları nasıl işleyebiliriz? Çünkü bu durumda o pencerenin pencere fonksiyonu bizim elimizde olmayacaktır. İşte standart pencereleri oluşturan Microsoft o pencerelerde bazı eylemler gerçekleştiğinde SendMessage yoluyla o standart pencerenin üst penceresine WM\_COMMAND isimli bir mesaj göndermektedir. Böylece biz üst pencerede WM\_COMMAND mesajı yoluyla alt penceredeki bazı mesajları alıp işleyebilmekteyiz.

WM\_COMMAND mesajında HIWORD(wParam) alt pencereden gelen mesajın türünü, LOWORD(wParam) ise alt pencere “id değeri”ni belirtmektedir. Windows’ta handle değerlerinin yanı sıra alt pencerelerin bir de “id değerleri” vardır. Alt pencere id değerleri alt pencereleri birbirlerinden ayırmak için kullanılır. Bu nedenle bunlar sistem genelinde tek değildir, kardeş pencereler temelinde tektir. Alt pencerelerin id değerleri CreateWindow fonksiyonunda HMENU parametresi yoluyla girilir. Bunların dışında WM\_COMMAND mesajındaki lParam parametresi de alt pencerenin handle değerini bulundurmaktadır. Aşağıda alt pencere mesajlarının işlenmesine yönelik bir örnek göreceksiniz.

#include <stdio.h>

#include <windows.h>

#define ID\_BUTTONOK 100

#define ID\_BUTTONCANCEL 101

#define ID\_EDITNAME 102

#define ID\_EDITNO 103

LRESULT CALLBACK WndProcParent(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam);

LRESULT CALLBACK WndProcChild(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam);

HWND g\_hWndParent;

HWND g\_hButtonOk;

HWND g\_hButtonCancel;

HWND g\_hEditBoxName;

HWND g\_hEditBoxNo;

HWND g\_hStaticName;

HWND g\_hStaticNo;

HINSTANCE g\_hInstance;

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpszCmdParam, int nCmdShow)

{

WNDCLASS wndClassParent;

MSG message;

if (!hPrevInstance) {

wndClassParent.style = CS\_HREDRAW | CS\_VREDRAW;

wndClassParent.cbClsExtra = 0;

wndClassParent.cbWndExtra = 0;

wndClassParent.hInstance = hInstance;

wndClassParent.hIcon = LoadIcon(NULL, IDI\_QUESTION);

wndClassParent.hbrBackground = GetStockObject(WHITE\_BRUSH);

wndClassParent.hCursor = LoadCursor(NULL, IDC\_ARROW);

wndClassParent.lpszMenuName = NULL;

wndClassParent.lpszClassName = "GenericParent";

wndClassParent.lpfnWndProc = (WNDPROC)WndProcParent;

if (!RegisterClass(&wndClassParent))

return -1;

}

g\_hInstance = hInstance;

g\_hWndParent = CreateWindow("GenericParent", "Sample Windows", WS\_OVERLAPPEDWINDOW, CW\_USEDEFAULT, 0,

450, 200, NULL, NULL, hInstance, NULL);

if (!g\_hWndParent)

return -1;

ShowWindow(g\_hWndParent, SW\_RESTORE);

UpdateWindow(g\_hWndParent);

while (GetMessage(&message, 0, 0, 0)) {

TranslateMessage(&message);

DispatchMessage(&message);

}

return message.wParam;

}

LRESULT CALLBACK WndProcParent(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)

{

switch (message) {

case WM\_CREATE:

g\_hButtonOk = CreateWindow("button", "Ok", WS\_CHILD | WS\_BORDER | WS\_VISIBLE | BS\_PUSHBUTTON , 260, 120,

70, 25, hWnd, (LPVOID)ID\_BUTTONOK, g\_hInstance, NULL);

g\_hButtonCancel = CreateWindow("button", "Cancel", WS\_CHILD | WS\_BORDER | WS\_VISIBLE | BS\_PUSHBUTTON, 340, 120,

70, 25, hWnd, (LPVOID)ID\_BUTTONCANCEL, g\_hInstance, NULL);

g\_hStaticName = CreateWindow("static", "Adı Soyadı", WS\_CHILD | WS\_VISIBLE, 10, 10,

100, 20, hWnd, NULL, g\_hInstance, NULL);

g\_hStaticNo = CreateWindow("static", "No", WS\_CHILD | WS\_VISIBLE, 10, 60,

100, 20, hWnd, NULL, g\_hInstance, NULL);

g\_hEditBoxName = CreateWindow("edit", "", WS\_CHILD | WS\_BORDER | WS\_VISIBLE, 10, 30,

200, 20, hWnd, (LPVOID)ID\_EDITNAME, g\_hInstance, NULL);

g\_hEditBoxNo = CreateWindow("edit", "", WS\_CHILD | WS\_BORDER | WS\_VISIBLE, 10, 80,

200, 20, hWnd, (LPVOID)ID\_EDITNO, g\_hInstance, NULL);

break;

case WM\_COMMAND:

switch (LOWORD(wParam)) {

case ID\_BUTTONOK:

if (HIWORD(wParam) == BN\_CLICKED) {

char name[120];

char no[120];

char text[240];

GetWindowText(g\_hEditBoxName, name, 120);

GetWindowText(g\_hEditBoxNo, no, 120);

sprintf(text, "Adı Soyadı: %s, No: %s", name, no);

MessageBox(hWnd, text, "Mesaj", MB\_OK);

}

break;

case ID\_BUTTONCANCEL:

if (HIWORD(wParam) == BN\_CLICKED) {

MessageBox(hWnd, "Cancel tuşuna tıklandı!", "Mesaj", MB\_OK);

}

break;

}

break;

case WM\_LBUTTONDOWN:

MessageBox(hWnd, "Parent WM\_LBUTTONDOWN", "Message", MB\_OK);

break;

case WM\_DESTROY:

PostQuitMessage(0);

break;

default:

return DefWindowProc(hWnd, message, wParam, lParam);

}

return 0;

}

**1.5. XWindow (X11) Sistemlerinde GUI Programlama Modelinin Temelleri**

Bu bölümde XWindow sistemlerindeki GUI programlama modelinin ayrıntılarına Windows’taki kadar girmeyeceğiz. Yalnızca bir “Merhaba Dünya” programı eşliğinde bazı temel açıklamaları yapmakla yetineceğiz. Önceki bölümlerde de belirttiğimiz gibi bu sistemlerde çeşitli düzeylerde uygulama yapabilmek için kütüphaneler ve ortamlar (frameworks) bulunmaktadır. Ancak XWindow sistemlerinin aşağı seviyeli çalışmasını en iyi betimleyen kütüphaneler XLIB ve XCB kütüphaneleridir. Biz burada yalnızca temel bir XLIB örneği vereceğiz.

**1.5.1. XWindow Sistemleri İçin XLIB Fonksiyonlarıyla “Merhaba Dünya” Programı**

Daha önceden de belirttiğimiz gibi XWindow (X11) sistemleri client-server biçimde çalışmaktadır. Burada biz bu sistemlerde boş bir pencere çıkartan örnek bir program vereceğiz. XLIB seviye olarak yukarıda gördüğümüz Windows API sistemine göre biraz daha aşağı seviyeli gibi durmaktadır. XLIB’i kullanan GTK+ kütüphanesinin seviye olarak Windows API fonksiyonlarına daha çok benzediğini söyleyebiliriz. Aşağıda ekrana bir ana pencere çıkartan örnek bir XLIB programı görüyorsunuz:

/\* xlib-helloworld.c \*/

#include <X11/Xlib.h>

#include <stdio.h>

#include <stdlib.h>

int main(void)

{

Display \*disp;

Window w;

XEvent e;

int scr;

disp = XOpenDisplay(NULL);

if (disp == NULL) {

fprintf(stderr, "Cannot open display\n");

exit(1);

}

scr = DefaultScreen(disp);

w = XCreateSimpleWindow(disp, RootWindow(disp, scr), 10, 10, 100, 100, 1,

BlackPixel(disp, scr), WhitePixel(disp, scr));

XSelectInput(disp, w, ExposureMask | KeyPressMask);

XMapWindow(disp, w);

for (;;) {

XNextEvent(disp, &e);

if (e.type == KeyPress)

break;

}

XCloseDisplay(disp);

return 0;

}

Merhaba dünya programındaki XOpenDisplay fonksiyonu XWindow sunucusu ile bağlantı yapmak için kullanılmaktadır. Bu fonksiyon başarı durumunda bize bir “Display handle” verir. Daha sonra biz bu handle’ı vererek bir ekran (screen) elde ederiz. Bu işlem DeafultScreen fonksiyonuyla yapılmaktadır. Örnek programımızda daha sonra ana pencere XCreateSimpleWindow fonksiyonuyla yaratılmıştır. Bu fonksiyon bize yaratılan pencereye ilişkin bir handle değerini Window \* türü olarak vermektedir. Programda daha sonra mesaj döngüsüne girmeden önce hangi girdi olaylarının izleneceğini belirlemek için XSelectInput fonksiyonu çağrılmıştır. Mesaj döngüsünden sıradaki mesaj XNextEvent fonksiyonuyla elde edilmektedir. Bu fonksiyon bize kuyruktaki mesajı XEvent isimli bir yapı olarak verir. Örnek programımızda bir tuşa basıldığında mesaj döngüsünden çıkılmaktadır. Mesaj döngüsünden çıkıldığında XCloseDisplay fonksiyonu ile daha önce alınmış olan ekran geri bırakılmıştır. Tabii ekran yok edildiğinde tüm pencereler de yok edilecektir. Ayrıca program sonlandığında X11 sistemi ile bağlantı da otomatik koparılmaktadır.

Bu bölümde XLIB programlamasına ilişkin başka ayrıntı verilmeyecektir. XLIB ile ilgili çeşitli kitaplardan ve dokümanlardan bilginizi ilerletilebilirsiniz. XLIB programları derlenirken libX11 kütüphanesinin link aşamasına dahil etmeyi unutmayınız. Derleme işlemi aşağıdaki gibi yapılabilir:

gcc -o xlib-helloworld xlib-helloworld.c -lX11

**2. Proseslerin Heap Alanları ve Tahsisat Algoritmaları**

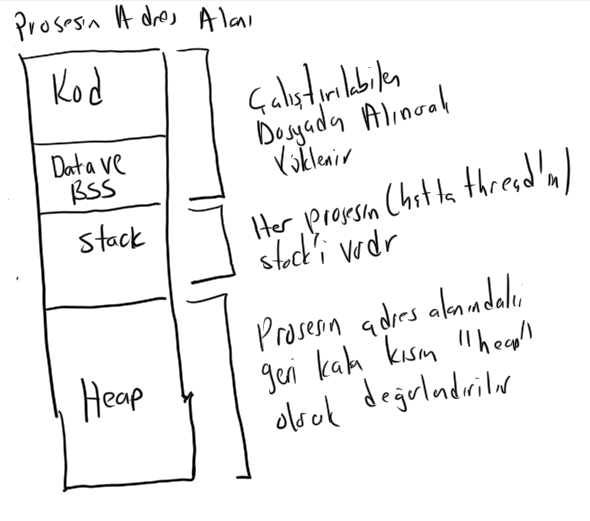
Programlama dillerinde programın çalışma zamanı sırasında bellekte tahsisat yapan çeşitli fonksiyonlar bulunabilmektedir. Örneğin C’deki malloc, calloc, realloc ve free fonksiyonları, C++’taki operator new ve operator delete fonksiyonları, C# ve Java’daki new operatörleri programın çalışma zamanı sırasında tahsisat yapma amacıyla kullanılırlar. Genel olarak programlama dillerinde dinamik bellek tahsisatları için kullanılan potansiyel bellek alanına “heap” denilmektedir. Tabii “heap” genel bir kavramdır. Herhangi bir sistem belli bir bellek aalanını heap olarak olarak belirleyip bir veri yapısı eşliğinde orayı dinamik tahsisatlar için kullanabilir. Bu bölümde dinamik tahsisatların gerçekleştirilmesinde kullanılan veri yapıları ve algoritmalar üzerinde durulacaktır. Yani bu bölümde, “tahsisat algoritmaları bir yerin tahsis edilip edilmediğini nasıl belirlemektedir? Alan serbest bırakıldığında aslında neler yapılmaktadır? Tahsisat sistemlerinin performanslarına neler etki etmektedir?” gibi soruların yanıtları uygulamalı olarak ele alınacaktır.

Tahsisat algoritmalarının işletim sistemlerinin gerçekleştirilmesinde de önemli bir yeri vardır. Çünkü işletim sistemleri de çeşitli çekirdek (kernel) nesnelerini kendi oluşturdukları bir heap sistemi içerisinde tahsis ederler. Örneğin bir proses yaratıldığında Linux’ta proses kontrol bloğunu temsil eden bir task\_struct yapısı oluşturulmaktadır. İşte bu task\_struct yapısı çekirdeğin kendi heap alanından “dilimli tahsisat sistemi (slab allocator)” denilen bir dinamik tahsisat yöntemiyle tahsis edilmektedir. Benzer biçimde Linux’ta her dosya açıldığında çekirdek alanında dosya nesnesini temsil file isminde yapı tahsis edilir. Bu tahsisat da yine çekirdeğin heap alanında yapılmaktadır. Bunun gibi işletim sisteminin çekirdeğinde onlarca türden nesneler gerektiğinde tahsis edilip geri bırakılmaktadır.

**2.1. Proseslerin Heap Alanları**

Modern ve kapasiteli işletim sistemlerinde heap alanı genellikle proses temelinde oluşturulmaktadır. Yani klasik masaüstü ve mobil işletim sistemlerinde her prosesin ayrı bir heap alanı vardır. Proses yaratıldığında işletim sistemi proesin bellek alanında o proses için bir heap alanı da oluşturur. Proses sonlandığında prosesin bellek alanı boşaltılırken heap alanı da yok edilmektedir. Windows sistemlerinde, UNIX/Linux sistemlerinde ve Mac OS X sistemlerinde, Android ve IOS gibi mobil işletim sistemlerindeki heap kullanımı hep böyledir. Proseslerin heap alanlarının yönetilmesi pek çok sistemde çekirdeğin bir fonksiyonu değildir. Bu yönetim kullanıcı seviyesindeki (user level) kodlarla yapılmaktadır. Bu organizasyonları malloc, calloc, realloc ve free gibi kullanıcı düzeyindeki (user level) kütüphane fonksiyonları yapmaktadır.

Genel olarak bir proses yaratıldığında prosesin potansiyel kullanımına ayrılan belleğe “prosesin adres alanı (process address space)” denilmektedir. Prosesin adres alanının önemli bir kısmı çalıştırılabilen (executable) dosyadan alınarak oluşturulmaktadır. Örneğin Windows’un PE (Portable Executable) formatında ve UNIX/Linux sistemlerinde kullanılan ELF (Executable and Linkable Format) formatında çalıştırılabilen dosyalar bölümlerden (sections) oluşur. Programın tüm makine kodları çalıştırılabilen dosyanın “kod bölümünde”, tüm global değişkenler “data” ve “bss” bölümlerinde bulunular. İşletim sisteminin yükleyicisi program için bir de stack alanı tahsis eder. İşte bu alanların dışında genellikle işletim sistemlerinde adres prosesin adres alanının geri kalan kısmı “heap” olarak düzenlenmektedir.



Yukarıdaki şekil kavramsal olarak çizilmiştir. Belli bir işletim sisteminde bu konuya ilişkin çeşitli ayrıntılar bulunmaktadır. Genel olarak heap alanının yerinin ve uzunluğunun sistemden sisteme değişebildiğini söyleyebiliriz.

Heap ile ilgili sıkça sorulan sorular şunlardır:

**Soru:** Heap alanı nerede oluşturulmaktadır?

**Yanıt:** Heap alanı prosesin adres alanı içerisinde oluşturulur. Yeri sistemden sisteme değişebilmektedir. Genellikle prosesin adres alanındaki geri kalan türm bölgeler heap olarak kullanılmaya uygundur.

**Soru:** Heap alanın büyüklüğü ne kadardır?

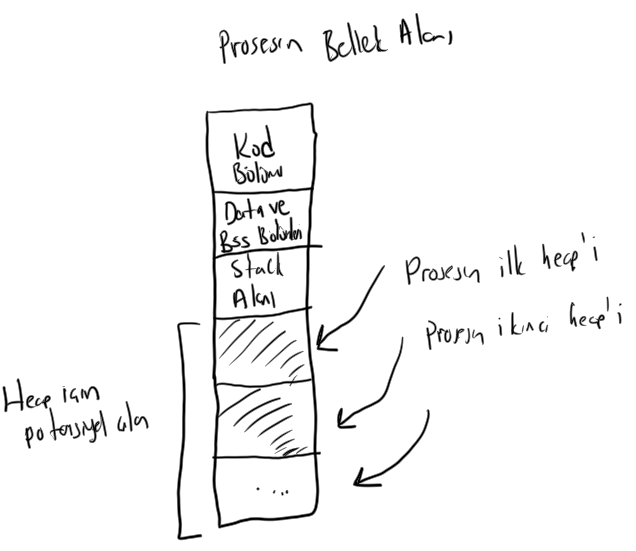
**Yanıt:** Heap alanının büyüklüğü sistemden sisteme değişebilmektedir. Bu büyüklük tabii prosesin o sistemdeki adres alanıyla da ilgilidir. Örneğin 32 bit Windows sistemlerinde prosesler zaten en fazla 2GB bellek alanına sahip olabilirler. Bu 2 GB’nin içerisinde “.text”, “.data”, “.bss” ve stack alanları da dahildir. O halde bu sistemlerde heap zaten en fazla 2 GB olabilir.

**Soru:** Heap alanı ortak bir alan mıdır, prosese özgü müdür?

**Yanıt:** Bu durum sistemden sisteme değişebilir. Ancak yaygın sistemlerin hemen hepsinde heap alanı prosesin adres alanı içerisinde proses özgü bir biçimde oluşturulmaktadır. Dolayısıyla proses sonlandığında o prosesin heap alanı da yok edilmektedir. Heap alanı bu sistemlerde thread’ler atarfından ortak erişilen bir alandır. Yani her thread’in ayrı bir heap alanı yoktur.

**2.2. Windows Sistemlerinde Heap Organizasyonu**

Windows sistemlerinde prosesler tek bir heap alanına sahip olmak zorunda değildir. Proses kendisi için birden fazla heap alanı yaratabilir.



Heap alanlarının yaratılması ve o alanlar üzerinde tahsisatların yapılabilmesi için HeapXXX biçiminde isimlendirilmiş API fonksiyonları bulunmaktadır. <bunların prototiplerini aşağıda görüyorsunuz:

HANDLE WINAPI HeapCreate(

\_\_in DWORD flOptions,

\_\_in SIZE\_T dwInitialSize,

\_\_in SIZE\_T dwMaximumSize

);

BOOL WINAPI HeapDestroy(

\_\_in HANDLE hHeap

);

LPVOID WINAPI HeapAlloc(

\_\_in HANDLE hHeap,

\_\_in DWORD dwFlags,

\_\_in SIZE\_T dwBytes

);

LPVOID WINAPI HeapReAlloc(

\_\_in HANDLE hHeap,

\_\_in DWORD dwFlags,

\_\_in LPVOID lpMem,

\_\_in SIZE\_T dwBytes

);

BOOL WINAPI HeapFree(

\_\_in HANDLE hHeap,

\_\_in DWORD dwFlags,

\_\_in LPVOID lpMem

);

HeapCreate fonksiyonu belli uzunlukta yeni bir heap alanı yaratır ve yaratılan heap alanının handle değeri ile geri döner. Yaratılan bir heap HeapDestroy fonksiyonuyla yok edilebilir. HeapAlloc fonksiyonu belli bir heap’ten tahsisat yapmaktadır. HeapReAlloc tahsis edilen alanın büyütülmesi ya da küçültülmesi için kullanılır. HeapFree ise alanı serbest bırakır.

Windows’ta bir proses yaratılırken işletim sistemi o proses için başlangıçta bir heap de yaratmaktadır. Buna prosesin default heap’i denir. Prosesin default heap’inin uzunluğu çalıştırılabilen dosyanın (PE dosyasının) içerisine bağlayıcı (linker) tarafından yazılmaktadır. Bu uzunluk bağlayıcı ayarları ile belirlenebilir. Fakat Microsoft bağlayıcıları aksi belirtilmediği durumda bu uzunluğu default 1 MB olarak almaktadır. Prosesin default heap’ine ilişkin handle değeri GetProcessHeap API fonksiyonuyla elde edebilir:

HANDLE WINAPI GetProcessHeap(void);

Windows’ta örnek bir heap yaratarak ondan tahsisat yapan yalın bir program şöyle yazılabilir:

#include <stdio.h>

#include <stdlib.h>

#include <windows.h>

int main(void)

{

HANDLE hHeap;

char \*str;

if ((hHeap = HeapCreate(0, 0x100000, 0x100000)) == NULL) {

fprintf(stderr, "cannot create heap!..\n");

exit(EXIT\_FAILURE);

}

if ((str = (char \*)HeapAlloc(hHeap, 0, 100)) == NULL) {

fprintf(stderr, "cannot allocate memory!..\n");

exit(EXIT\_FAILURE);

}

strcpy(str, "This is a test");

puts(str);

HeapFree(hHeap, 0, str);

HeapDestroy(hHeap);

return 0;

}

Bu örnekte proses için 1 MB uzunluğunda bir heap yaratılmıştır, sonra o heap’ten tahsisat yapılmıştır. Proses bittiğinde prosesin tüm heap alanları zaten yok edilmektedir.

Prosesin default heap’i proses yaratıldığında otomatik oluşturulduğu için hemen tahsisat işleminde kullanılabilir. Örneğin:

#include <stdio.h>

#include <stdlib.h>

#include <windows.h>

int main(void)

{

HANDLE hHeap;

char \*str;

if ((hHeap = GetProcessHeap()) == NULL) {

fprintf(stderr, "cannot get default heap handle!..\n");

exit(EXIT\_FAILURE);

}

if ((str = (char \*)HeapAlloc(hHeap, 0, 100)) == NULL) {

fprintf(stderr, "cannot allocate memory!..\n");

exit(EXIT\_FAILURE);

}

strcpy(str, "This is a test");

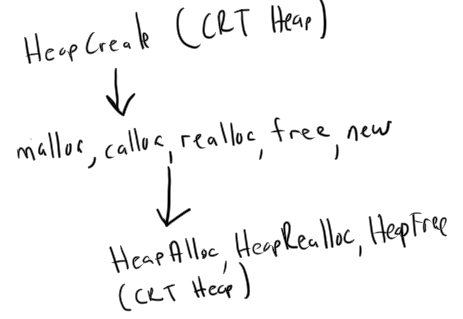
puts(str);

HeapFree(hHeap, 0, str);

return 0;

}

Pekiyi Windows sistemlerinde bizim C’de kullandığımız malloc, calloc, realloc ve free fonkisyonları, C++’taki operator new ve operator delete fonksiyonları arka planda hangi heap’ten tahsisat yapmaktadır? İşte akış main fonksiyonuna gelmeden önce derleyicinin yerleştirdiği başlangıç kodu (start up module) yoluyla ismine “CRT (C Runtime) Heap” denilen bir heap yaratılmaktadır. malloc, calloc ve realloc gibi standart C fonksiyonları bu heap’ten tahsisat yapmaktadır.



Pekiyi Windows sistemlerinde prosesin birden fazla heap’e sahip olmasının anlamı nedir? Heap alanında çok fazla tahsisat ve boşaltım yapıldığında “bölünme durumu (fragmentation)” oluşabilir. Bölünme ardışıl bellek miktarının azalmasına dolayısıyla da heap verimininin düşmesine yol açan bir olgudur. İşte birden fazla heap kullanımında bir heap’te bölünme olsa bile bu bölünme diğer heap’leri etkilememektedir. Yani heap’lerin birbirlerinden ayrılması bölünme olgusunu azaltıcı bir unsur oluşturabilmektedir.

Windows’ta Heap Kullanımına ilişkin sıkça sorulan sorular şunlar olabilir:

**Soru:** Windows’ta bir prosesin kaç heap’i vardır?

**Yanıt:** Windows’ta proses başlatıldığında default bir heap yaratılır. Ancak programcı isterse birden fazla heap yaratıp tahsisatlarını onların herhangi birinden yapabilir.

**Soru:** Windows’ta Microsoft’un C kütüphanesindeki malloc, calloc, realloc ve free fonksiyonları prosesin hangi heap’inden tahsisat yapmaktadır?

**Yanıt:** Bu fonksiyonlar derleyicilerin başlangıç kodları (start up code) tarafındna yaratılmış olan bir heap’ten (buna CRT heap deniyor) tahsisat yaparlar. (Tabii prosesin default heap’i de bu amaçla kullanılabilirdi. Ama Microsoft bu fonksiyonlar için ayrı bir heap yaratmayı uygun görmüştür.)

**Soru:** Prosesin default heap’inin uzunluğı nedir?

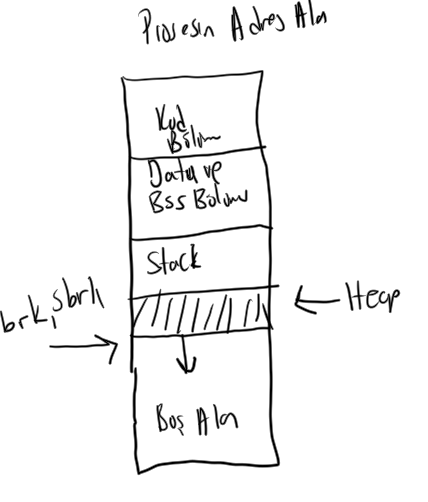
**Yanıt:** Bu uzunluk bağlayıcı ayarlarından değiştirilebilmekle birlikte bağlayıcı için default değer 1 MB’dir.

**Soru:** HeapCreate ile belli uzunlukta heap yaratmış olalım. Bu heap otomatik büyütülebilir mi?

**Yanıt:** Evet böyle bir olanak vardır. HeapCreate fonksiyonuyla heap alanı otomatik büyütülecek biçimde yaratılabilir. Örneğin prosesin default heap’i de CRT heap de böyle yaratılmışlardır.

**2.2. UNIX/Linux Sistemlerindeki Heap Organizasyonu**

UNIX/Linux sistemlerinde prosesin tek bir heap alanı vardır. Prosesin adres alanı başlangıçta çalıştırılabilen dosyanın içerisindeki bilgilerden oluşturulur. Fakat brk ve sbrk fonksiyonlarıyla prosesin adres alanı büyütülebilmektedir. İşte malloc, calloc, realloc fonksiyonları heap alanı yetmediğinde bu brk ve sbrk fonksiyonlarını kullanarak prosesin adres alanını büyütürler.



**2.3. Tahsisat Algoritmaları**

Belli bir alanın heap olarak kullanılabilmesi için o alanın bir veri yapısı ile organize edilmesi gerekir. Örneğin malloc gibi bir fonksiyon heap’te bir alanı tahsis ettiğinde o alanın tahsis edilmiş olduğunu bir biçimde bilmektedir. Böylece yeni bir malloc çağrısı bize aynı alanı vermez. free fonksiyonu da alanı iade ettiğinde artık o alan boş olarak işaretlenmektedir. Pekiyi belli bir alan üzerinde tahsisat yapıp onları geri bırakan bir veri yapısı nasıl oluşturulmaktadır?..

Tahsisat sistemlerinde kullanılan veri yapıları ve algoritmaların bilinmesinin faydaları şunlar olabilir:

- Belli bir sistem için biz bir heap sistemini oluşturmak isteyebiliriz. Bu durumda bu sistemlerin nasıl oluşturulduğunu bilmemiz gerekir. Örneğin bir işletim sistemi yazacak olalım. İşletim sisteminin çekirdeğinin kullandığı bir heap sisteminin olması gerekir değil mi? Bu durumda işletim sistemini gerçekleştiren kişilerin bu tahsisat sistemlerini biliyor olmaları gerekir. Benzer biçimde biz heap sistemi olmayan küçük bir mikrodenetleyicide çalışıyor olabiliriz. Onun için bir heap sistemi oluşturmak isteyebiliriz.

- Tahsisat sistemlerinde kullanılan veri yapılarının ve algoritmaların bilinmesi bu sistemlerin daha iyi analiz edilmesini ve değerlendirilmesini sağlayabilir. Bu bilgi bizim bilinç düzeyimizi yükseltebilir. Bazı olayları daha iyi anlamamız için katkı oluşturabilir.

İyi bir tahsisat sisteminin performans ölçütleri ne olabilir? Yani örneğin iki ayrı sistemde iki malloc, realloc ve free fonksiyonları bulunuyor olsun. Bunlardan hangisinin daha iyi olduğuna yönelik ölçütler nelerdir? İşte tahsisat sistemlerinin performans ölçütlerini üç maddede özetleyebiliriz:

1) Hız en önemli ölçütlerden biridir. Örneğin iki farklı malloc gerçekleştirimi olsun. Bunlardan hangisi daha hızlı bize bir boş alan vermektedir? İki free fonklsiyonundan hangisi alanı daha çabuk serbest bırakmaktadır?

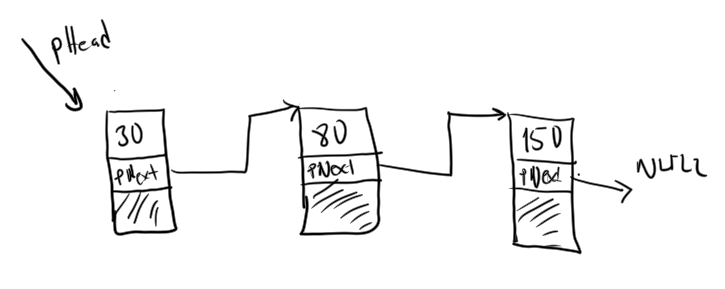
2) Bölünme (fragmentation) de önemli bir ölçüttür. Heap alanını daha az bölen, başka bir deyişle ardışıl bellek miktarını daha yüksek tutan sistemler diğerlerine tercih edilir.

3) Heap organizasyonu için oluşturulan veri yapısının kapladığı alanın (meta data alanının) küçüklüğü de bir performans ölçütüdür. Örneğin bir tahsisat alanını organize etmek için çok büyük bir alan kullanmak istemeyiz.

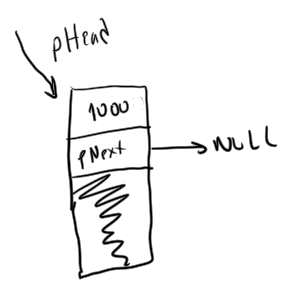
**2.3.1. Klasik Yöntem: Boş Blokların Bağlı Listede Tutulması**

Boş blokların bağlı listede tutulması en çok kullanılan tahsisat sistemi gerçekleştirim yöntemidir. Knuth bu yöntemi “The Art Of Computer Programming” kitabının birinci cildinde ele alınmıştır. Ayrıca Ritchie ve Kernigan’ın “The C Programming Language” kitabında da bu yöntem açıklanmaktadır. Bugün Windows sistemlerinde (yani HeapXXX fonksiyonlarında) ve UNIX/Linux sistemlerindeki C kütüphanelerinde temelk olarak bu yöntem kullanılmaktadır.

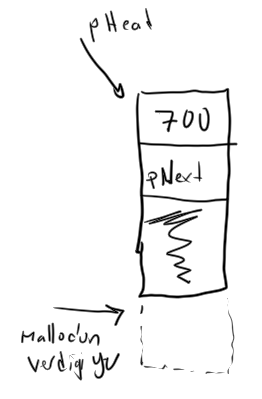
Bu yöntemde heap’teki boş alanlar bir bağlı listede tutulur. Bu boş alanların başında bir başlık kısmı vardır. Bu başlık kısmında bağlı listenin sonraki düğümünün yeri (next göstericisi) ve boş bloğun uzunluğu tutulmaktadır. Bu bağlı listeyi aşağıdaki şekille temsil edebiliriz:



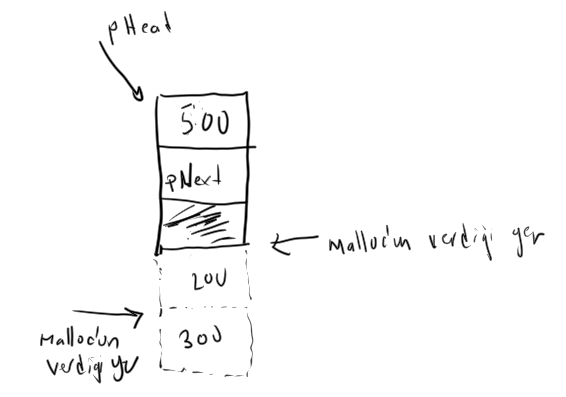
Bu şekilde bağlı listede sırasıyla 30 byte, 80 byte ve 150 byte boş blokların olduğunu görüyorsunuz. Pekiyi işin başında bu bağlı listenin durumu nasıldır? Örneğin heap’imizin 1000 byte olduğunu düşünelim. İşte işin başında bu listede tüm boş alanı tutan yalnızca tek bir eleman vardır:



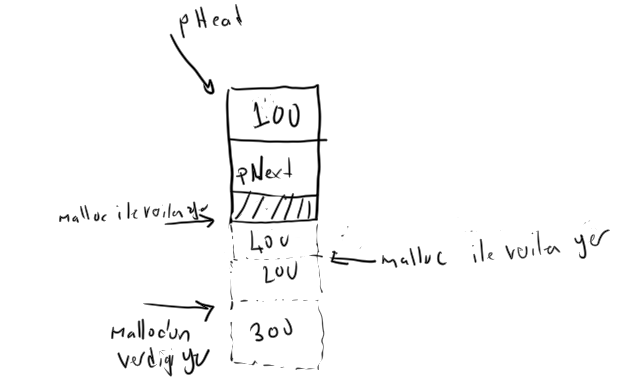
Sonra malloc gibi tahsisat fonksiyonları bu boş blok listesinde arama yaparak bize uygun bir blok verir. Bu yöntemde tahsis edilen blokların yerleri herhangi bir listede tutulmamaktadır. Yalnızca boş blokların kaydı bir bağlı listede tutulmaktadır. Örneğin böyle bir sistemde biz 300 byte tahsis etmiş olalım. Şimdi boş blok listesi şu hale gelecektir:



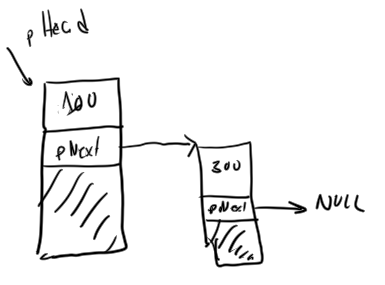
Şimdi de 200 byte daha tahsis etmiş olalım:



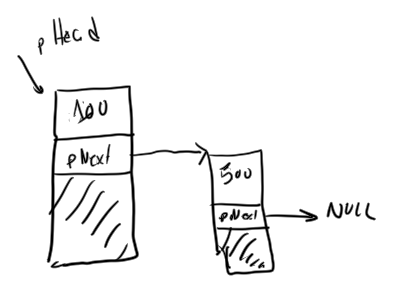
Şimdi 400 byte’lık bir tahsisat daha yapılmış olsun:



Şimdi 300’lük bloğun serbest bırakıldığını (free edildiğini) düşünelim. Artık bağlı listede iki eleman bulunacaktır:

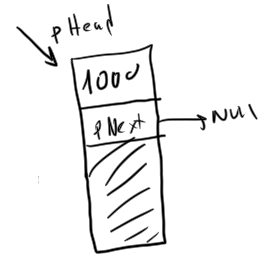


Tabii serbest bırakma işlemi yapılırken eğer serbest bırakılacakl blokla bağlı listedeki boş bir blok peşi sıra geliyorsa bunu ayrı blok olarak bağlı listede saklamak yerine bunları birleştirerek tek bir blok haline getirmek daha uygun olur. (Bunun nedenini düşününüz). Böylece ardışıl bloklar birleştirildiği için bağlı listedeki boş blokların hiçbiri ardışıl olmayacaktır. (Zaten ardışıl olanları birleştiriyoruz.) Örneğin şimdi biz daha önce tahsis etmiş olduğumuz 200 byte’lık bloğu serbest bırakmak isteyelim:



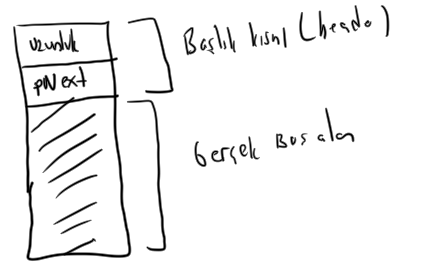
Gördüğünüz gibi 200 byte’lık blok 300 byte’lık blokla peşi sıra olduğu için birleştirilmiştir. Şimdi de 400 byte’lık bloğu serbest bırakalım:

,

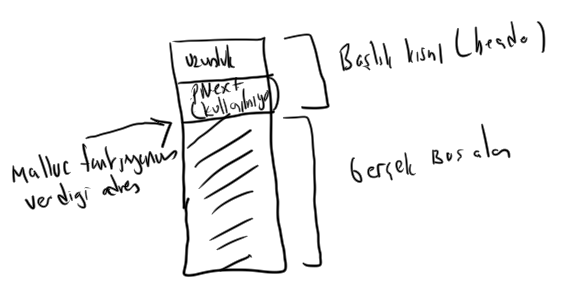


Gördüğünüz gibi tüm alanları serbest bırakınca ilk baştaki durumun aynısına geri döndük.

Şimdi veri yapısının ayrıntılarına girelim. Bu veri yapısında bağlı listedeki boş bloğun durumu şöyledir:



Gördüğünüz gibi boş bloğun başındaki başlık kısmında bloğun uzunluğu ve sonrakşi boş bloğun bağı (link) tutulmaktadır. Pekiyi tahsis edilmiş bir bloğun durumu nasıldır? Tahsistat fonksiyonu (örneğin malloc) bize boş blok listesinden uygun bloğu verirken o bloğun başlık kısmını ortadan kaldırmaz. Tahsisat fonksiyonunun bize verdiği adres bu başlık kısmından sonraki boş alanın adresidir. Dolayısıyla bize verilen adresten biraz yukarı çıktığımızda orada aslında orada tahsis edilen bloğun başlık kısmı vardır. O halde Tahsis edilmiş bloğun genel yapısı şöyledir:



Böylece bloğu serbest bırakan fonksiyon (örneğin free fonksiyonu) verilen adresin yukarısına bakarak tahsis edilmiş bloğun uzunluğunu görebilir. Tahsis edilmiş blok için artın bağ göstericisinin (next göstericisinin) bir anlamı kalmamıştır.

Pekiyi bu yöntemde tahsisat işlemini yapan fonksiyon nasıl başarısız oabilir mi? Evet, tahsisat fonksiyonu boş blok listesinde istenilen miktara eşit ya da ondan büyük hiç bir blok bulamazsa başarısız olacaktır.

Şimdi boş alanların bağlı listede tutulması tekniğinin gerçekleştirimini yapalım. Bu gerçekleştirimde heap olarak kullanacağımız alan bize başlangıç adresi ve uzunluğuyla verilmektedir. Biz de o alanı bu tekniğe göre organize edeceğiz. Gerçekleştirimde “handle tekniği” kullanılacaktır. Yani heap bilgileri bir yapıda tutulacak, onun başlangıç adresi bir handle olarak verilecek, bu handle kullanılarak da tahsisat işlemleri yapılacaktır.

Dinamik bellek tahsisatları genellikle byte düzeyinde belli bir değerin katlarında yaniş bir birim uzunlukl temelinde yapılmaktadır. Bu durum hizalama açısından fayda sağlamanın yanı sıra sistemin gerçekleştirimini de kolaylaştırmaktadır. Bizim gerçekleştirimimizde her boş bloğun başındaki başlık kısmının uzunluğu aynı zamanda birim uzunluk olarak alınacaktır. 32 bit sistemlerde bu birim uzunluk 8 byte’tır. Bu durumda örneğin biz 1 byte bile tahsis etmek istesek başlık kısmı haricinde aslında 8 byte’lık bir alan tahsis edilecektir.

Boş blokların başlık kısmı şağıdaki yapıyla temsil edilmektedir:

typedef struct tagBLOCK {

size\_t nunits;

struct tagBLOCK \*next;

} BLOCK;

#define UNIT\_SIZE sizeof(BLOCK)

Burada nunits başlık kısmı dahil olmak üzewre boş bloğun kaç birimden oluştuğunu belirtir. Bir birim sizeof(BLOCK) (yani 8 byte) uzunluğundadır.

Handle sistemi için aşağıdaki gibi bir yapı oluşturulabilir:

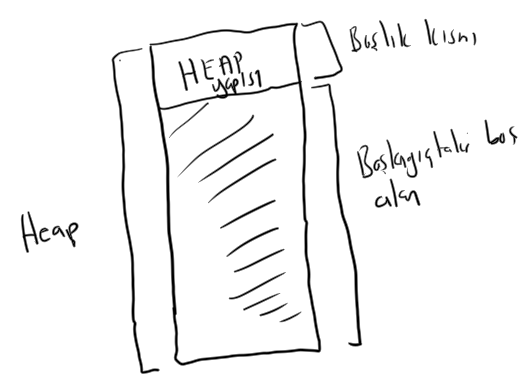
typedef struct tagHEAP {

BLOCK \*head;

size\_t sizeLeft; /\* byte cinsinden \*/

} HEAP, \*HHEAP;

Bu HEAP yapısı aslında heap olarak ayrılan alanın başındaki başlık “header” kısmı gibidir.



Gerçekleştirimimizde heap alanını yaratan fonksiyon CreateHeap isimli fonksiyondur:

HHEAP CreateHeap(void \*addr, size\_t size)

{

HHEAP hHeap;

BLOCK \*blk;

if (size < sizeof(HEAP) + UNIT\_SIZE)

return NULL;

hHeap = (HHEAP)addr;

hHeap->sizeLeft = size - sizeof(HEAP);

blk = (char \*)addr + sizeof(HEAP);

blk->nunits = hHeap->sizeLeft / UNIT\_SIZE;

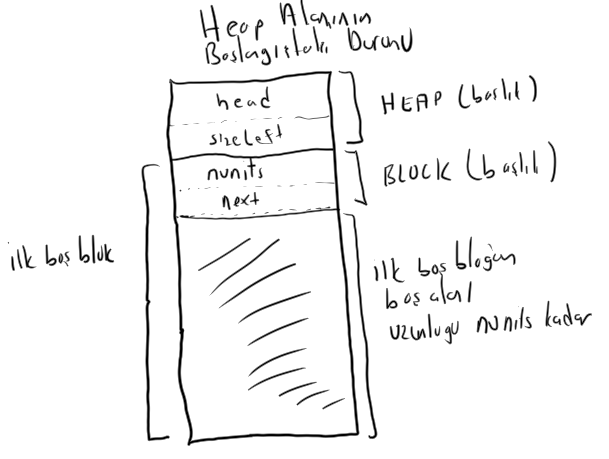
blk->next = NULL;

hHeap->head = blk;

return hHeap;

}

Burada heap olarak kullanılacak alanın başındaki HEAP yapısıyla temsil edilen başlık kısmı oluştuurlmuştur. Bu başlığın hemen altından başlayan geri kalan tüm alan başlangıçtaki boş bloktur.



Sistemimizde test amaçlı heap’in boş bloklarını listeleyen yardımcı bir fonksiyonun bulunması faydalı olacaktır:

void DispHeap(HHEAP hHeap)

{

BLOCK \*blk;

blk = hHeap->head;

printf("Free Heap Size (byte): %u\n", hHeap->sizeLeft);

while (blk != NULL) {

printf("Unit Number = %d, Block Unit Size = %u\n",

((char \*)blk - (char \*)hHeap->head) / UNIT\_SIZE, blk->nunits);

blk = blk->next;

}

printf("-------------------------------------\n");

}

Burada “Unit Number” heap’in hemen başındaki birim (yani 8 byte’lık tahsisat birimi) 0 olmak üzere ilgili birimin numarasını belirtmektedir. “Block Unit Size” ise boş bloğun kaç birimden oluştuğunu belirtir.

Şimdi en önemli fonksiyon olan tahsisat fonksiyonumuzu yazalım. Fonksiynumuzun ismi Malloc olsun:

void \*Malloc(HHEAP hHeap, size\_t size)

{

size\_t nunits;

BLOCK \*blk, \*prevBlk;

nunits = (size + UNIT\_SIZE - 1) / UNIT\_SIZE + 1;

blk = prevBlk = hHeap->head;

while (blk != NULL) {

if (blk->nunits >= nunits) {

if (blk->nunits == nunits) {

if (blk == hHeap->head)

hHeap->head = blk->next;

else

prevBlk->next = blk->next;

}

else {

blk->nunits -= nunits;

blk += blk->nunits;

blk->nunits = nunits;

}

hHeap->sizeLeft -= nunits \* UNIT\_SIZE;

return blk + 1;

}

prevBlk = blk;

blk = blk->next;

}

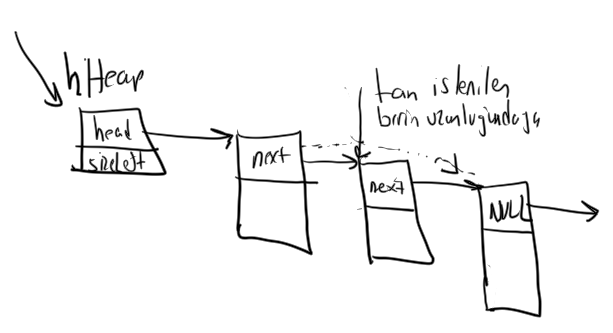
return NULL;

}

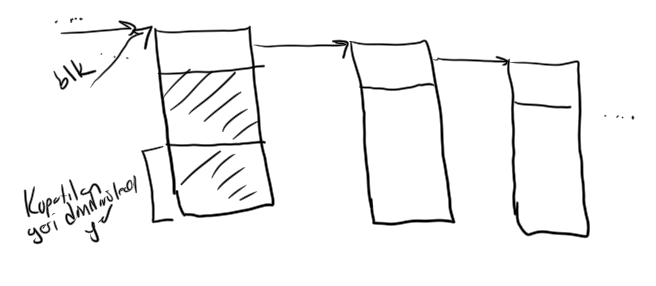
Burada önce tahsisat için gereken birim uzuluğu hesaplanmıştır:

nunits = (size + UNIT\_SIZE - 1) / UNIT\_SIZE + 1;

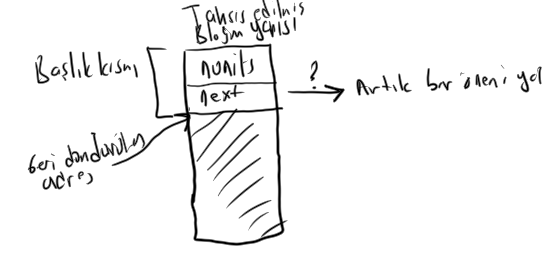
Daha sonra boş blok listesi bir döngü ile dolaşılmaktadır. Örneğimizdeki Malloc fonksiyonunda “First fit” algoritması uygulanmıştır. Bu durumda istenilen miktara eşit ya da ondan büyük olan ilk boş bloktan tahsisat yapılır. Bazı özel durumlara bakılmıştır. Örneğin boş blok tam olarak istenilen uzunluktaysa bu özel bir durumdur. Çünkü bu durmda önceki düğüm üzerinde değişikil yapılmalıdır:



Eğer boş blok büyükse o bloğun aşağısından gerekli boş alan kopartılmıştır.



Geri döndürülen adresin başlık kısmından bir birim ilerisi olduğuna dikkat ediniz. Böylece tahsis edilmiş olan bloğun hemen yukarısında bir başlık kısmı bulunmaktadır. Tabii bu başlık kısmının next göstericisinin bir önemi yoktur. Fakat birim uzunluğunun free işlemi için önemi varıdr:



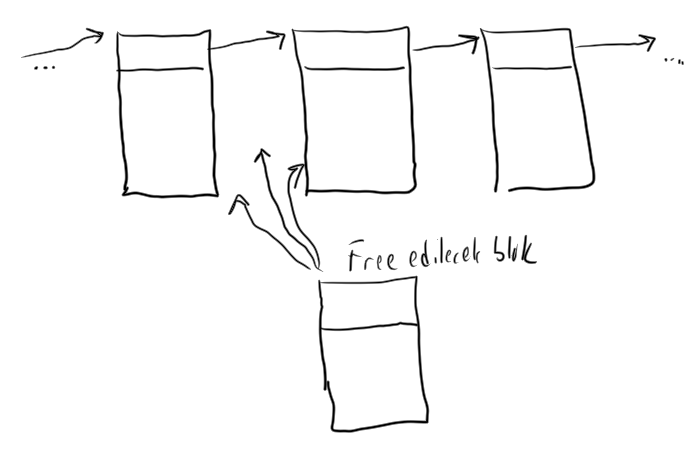
Şimdi sıra Free fonksiyonunu yazmaya geldi. Free fonksiyonunda dikkat eedilmesi gereken birkaç durum vardır:

1) Boş blokları tutan bağlı listedeki bloklar ardışıl değildir. Çünkü Free zaten bunları ardışıl hale getirecek biçimde yazılacaktır.

2) Free ile blok boş blok listesine hemen eklenmemeli. Bir birleştirme durumu var mı diye bakılmalıdır? Bölünme durumunu engellemek için mümkün olduğunca büyük blokların oluşturulması gerektiğini anımsayınız.

3) Boş blok listesindeki bloklar adrese göre sıralı durumdadır. Bu nedenle Free fonksiyonu hiç birleştirme yapmasa bile boş bloğu listenin sonuna değil adres sırasını bozmadan uygun yerine insert etmelidir.

Birleştirme için üç durum söz konusudur: Soldaki boş bloğun sağı ile birleştirme, Sağdaki boş bloğun solu ile birleştirme ve hem soldaki bloğun sağı hem de sağdaki bloğun solu ile tam birleştirme. Free algoritmasının bunu yapabilmesi gerekir.



Fonksiyon aşağıdaki gibi gerçekleştirilebilir:

void Free(HHEAP hHeap, void \*addr)

{

BLOCK \*blk, \*freeBlk;

blk = hHeap->head;

freeBlk = (BLOCK \*)addr - 1;

hHeap->sizeLeft += freeBlk->nunits \* UNIT\_SIZE;

if (hHeap->head == NULL) { /\* special case 1: is list empty?\*/

hHeap->head = freeBlk;

freeBlk->next = NULL;

return;

}

if (hHeap->head > freeBlk) { /\* special case 2: add as a first node \*/

if (freeBlk + freeBlk->nunits == hHeap->head) {

freeBlk->nunits += hHeap->head->nunits;

freeBlk->next = hHeap->head->next;

}

else

freeBlk->next = hHeap->head;

hHeap->head = freeBlk;

return;

}

while (blk->next != NULL && blk->next < freeBlk)

blk = blk->next;

if (freeBlk + freeBlk->nunits == blk->next) {

freeBlk->nunits += blk->next->nunits;

freeBlk->next = blk->next->next;

}

else

freeBlk->next = blk->next;

if (blk + blk->nunits == freeBlk) {

blk->nunits += freeBlk->nunits;

blk->next = freeBlk->next;

}

else

blk->next = freeBlk;

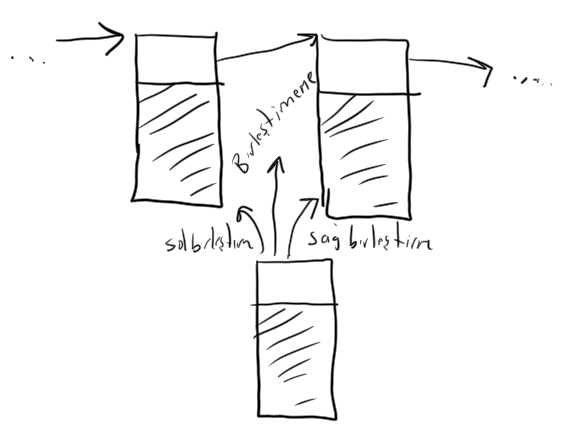
}

Burada önce iki özel duruma bakılmıştır:

1) Heap tamamen boş mudur durumu: Eğer böyleyse boş blok ilk blok olarak eklenip hHeap->head güncellenmiştir.

2) Ekleme hemen bağlı listenin başına mı yapılmaktadır? Bu durumda sağdaki düğümün solu ile birleştirme mümkünse yapılmış ve düğüm bağlı listenin başına eklenmiştir.

Gerçekleştirimde önce free edilecek bloğun bağlı listedeki yeri bulunmuştur. Sonra blok birleştirmesi durumuna bakılmıştır. Bunun için önce sağ tarafla birleştirme durumu kontrol edilmiştir. Sonra sol taraf ile birleştirmeye bakılmıştır.



Yukarıdaki gerçekleştirimde handle alanında (HEAP yapısında) yalnızca head göstericisi tutulmuştur. Eğer handle alaınında head göstericisi değil de bu niyetle BLOCK türünden bir nesne tutarsak özel durumlar ortadan kaldırılabilir ve özellikle Free algoritmaıs biraz da ha sade yazılabilir:

typedef struct tagHEAP {

BLOCK head; /\* gösterici değil kendisi \*/

size\_t sizeLeft; /\* byte cinsinden \*/

} HEAP, \*HHEAP;

Pekiyi boş listesinin çift bağlı liste (double linked list) olmasının bir faydası olabilir mi? Bilindiği gibi çift bağlı listeler iki amaçla tercih edilmektedir:

1) Tersten dolaşımı mümkün hale getirmek

2) Adresi bilinen bir düğümü sabit zaman karmaşıklıkta (O(1) karmaşıklıkta) silmek.

İşte çift bağlı liste yalnızca bizim tahsisat sırasında önceki düğümü tutmamızı engeller. Bu da ciddi bir fayda sağlamamaktadır. Üstelik çift bağlı listenin düğümleri (yani bloların başlıkları) daha fazla yer kaplayacaktır.

**2.3.2. İkiz Blok Tahsisat Sistemi (Buddy Allocator)**

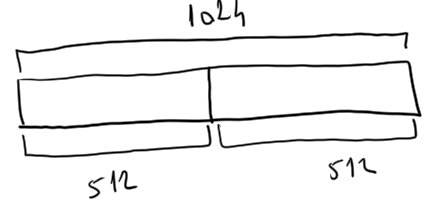
İkiz blok blok sistemi (buddy allocator) Linux başta olmak üzere bazı işletim sistemlerinin bellek yönetimlerinde kullanılmaktadır. Linux işletim sistemli sayfaları (page frames) bu tahsisat sistemine göre tahsis eder. Linux’un çekirdek heap sistemini oluşturan “dilimli tahsisat sistemi (slab allocator)” de ikiz blok sistemi üzerine oturtulmuştur.

İkiz blok sisteminin en önemli avantajları tahsisatların ve geri bırakmaların hızlı bir biçimde yapılmasıdır. Bu sistemde dışsal bölünme (external fragmentation) oldukça iyi bir düzeyde tutulur. Ancak içsel bölünme (internal fragmentation) dolayısıyla oluşan kayıplar önemli boyutlara varabilmektedir. Tipik olarak tahsis edilen alanların %20 civarı içsel bölünme nedeniyle harcanmaktadır. İkiz blok sistemini gerçekleştirmek için kullanılan “metadata” alanları boş blok listesi sistemini gerçekleştirmek için kullanılan “metadata” alanlarından kimi durumlarda daha büyük olabilmektedir. Ayrıca bu sistemin gerçekleştirimi boş blok listesi sistemine göre daha zordur.

İkiz blok sisteminde tahsis edilebilecek blokların uzunlukları 2n biçimindedir. Burada n değerine “mertebe (order)” denir. Sistemde genellikle başlangıçta 2n’lik tek bir blok bulunur (ancak başlangıçta tek bloğun bulunması zorunlu değildir). Başlangıçtaki bu n değerine maksimum mertebe (maximum order) denir. İkiz blok sisteminde tahsis edilebilecek en küçük mertebenin (minimum order) de başlangıçta belirlenmesi gerekir. Teorik olarak en küçük blok 20 = 1 byte olsa da, 1 byte tahsisat için çok küçük bir alandır. Örneğin en büyük mertebenin 10, en küçük mertebenin 3 olduğu bir sistem düşünelim. Bu durumda başlangıçtaki bellek miktarı 210 = 1024, tahsis edilebilecek en küçük blok uzunluğu da 23 = 8 olacaktır.



İkiz blok sisteminde yan yana iki 2k’lik 2k-1’lik bir blok biçiminde ele alınmaktadır. Örneğin 1024’lük blok yan yana 2 tane 512’lik blok gibi ele alınabilir:



İşte yan yana iki tane 2k‘lık blok biribirlerinin ikizi (buddy’si) durumundadır. Belli bir anda ikiz bloklar için tahsisat durumu üç biçimde olabilir:

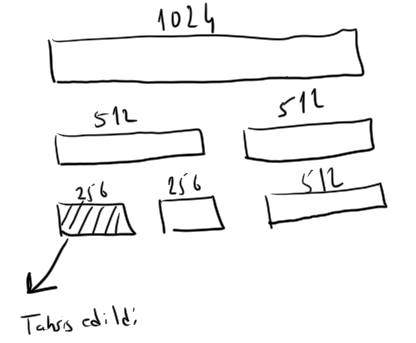
1) Bunlardan biri tahsis edilmiş biri edilmemiş olabilir.

2) Bunlardan her ikisi de tahsis edilmiş olabilir.

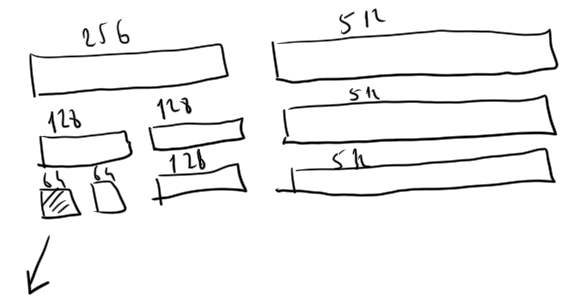
3) Bunların her ikisi de tahsis edilmemiş olabilir.

İşte ikizlerin her ikisi de tahsis edilmemiş duruma gelince bu bloklar birleştirilip daha yüksek mertebeye yükseltilmektedir. Bu durumda ikizlerden biri serbest bırakıldığında sistem diğer ikizin durumuna bakar; o da serbest bırakılmış durumdaysa onları birleştirir.

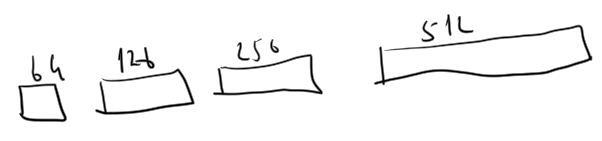
Yukarıda da belirttiğimiz gibi ikiz blok sisteminde genellikle başlangıçta tek bir blok vardır. Bir tahsisat yapılmak istendiğinde bu tek blok sürekli ikiye bölünerek istek karşılanmaya çalışılır. Örneğin maksimum mertebesi 10 olan (yani 1024 byte olan) bir sistemde 200 byte’lık bir alanın tahsis edilmek istendiğini düşünelim. İkiz blok sisteminde tahsis edilebilecek bloklar ikinin kuvvetleri uzunluğunda olmak zorundadır. Bu nedenle bu sistemde tam olarak 200 byte’lık bir blok tahsis edilemez. Onun yerine 200’den büyük olan 2’nin en küçük kuvveti kadar uzunlukta tahsisat yapılabilir. O da 256’dır. (Bu durumda 56 byte’ın içsel bölünme nedeniyle harcandığına dikkat ediniz.) Bu tahsisat sırasında bloklar ikiye bölünerek istek karşılanmaya çalışılır. Önce 1024’lük tek blok iki ayrı 512’lik blok olarak bölünür. Sonra bu 512’lik bloklardan biri de 256’lık iki blok olarak bölünür. Bu 256’lıklardan biri de tahsis edilir.



Böylece elimizde bir tane 512’lik bir tane de 256’lık blok kalmıştır. Şimdi 50 byte’lık bir bloğun daha tahsis edilmek istendiğini düşünelim. 50’ye en yakın 2’nin kuvveti 64’tür. Bu durumda 64 byte tahsis edilecektir. Tahsisat işlemi eldeki en küçük bloktan hareketle yürütülür. Elimizdeki en küçük blok 256’lıktır. O halde bu blok önce iki 128 olarak bölünür. Sonra o 128’lerden biri yine ikiye bölünerek istek karşılanır:



Şimdi elimizde bir tane 64’lük, bir tane 128’lik bir tane de 512’lik blok bulunmaktadır. Şimdi bu son durumda 256’lık bloğun serbest bırakıldığını düşünelim. Yeni durumdaki serbest bloklar şöyle olacaktır:

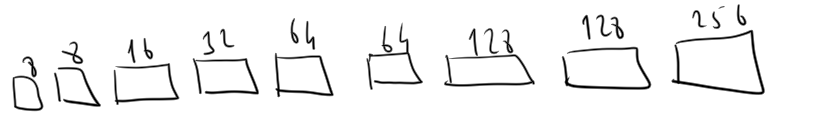


Bu sistemde nasıl tahsisat sırasında bloklar parçalanıyorsa serbest bırakma sırasında da mümkün olduğunca yan yaana bloklar birleştirilmeye çalışılır. Ancak örneğimizde 256’lık alanın serbest bırakılması herhangi bir birleştirmeye yol açmamaktadır. Çünkü birleştirme ancak ikizlerle (buddy’lerle) yapılır. Buradaki 256’lık bloğun ikizi tamamen serbest durumda değildir. Şimdi 64’lük bloğun da serbest bırakıldığını düşünelim. Bu 64’lük bloğun ikizi de serbest durumda olduğu için bunlar birleştirilir ve 128’lik tek blok haline getirilir. Bu 128’in de ikizi serbest olduğu için bunlar da birleştirilir ve 256’lık tek blok haline getirilir. Bu 256’nın da ikizi serbest durumda olduğu için bunlar da birleştirilir ve 512’lik tek blok oluşturulur. Nihayet bu 512’lik bloğun ikizi de serbest durumda olduğu için onlar da birleştirilecek ve 1024’lük tek blok elde edilecektir. Görüldüğü gibi ikiz blok sisteminde mümkün olduğunca ikizler birleştirilerek ardışıl büyük parçalar elde edilmeye çalışılmaktadır.

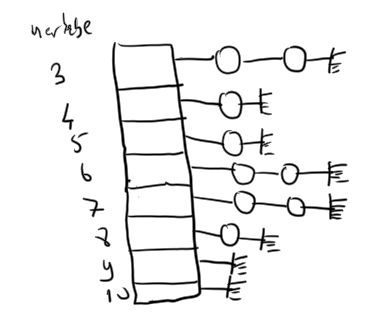
**2.3.2.1. İkiz Blok Sisteminin Gerçekleştirilmesi**

İkiz blok sistemi çok değişik biçimlerde gerçekleştirilebilmektedir. Ancak pek çok gerçekleştirim benzer veri yapılarından faydalanır. Gerçekleştirimde kullanılan veri yapılarının ana noktalı şunlardır:

1) Her mertebe için o mertebedeki boş blokları tutan bir bağlı liste bulundurulabilir. Örneğin maksimum mertebesi 10 olan ve minimum mertebesi 3 olan bir sistem için toplamda 10 – 3 + 1 = 8 tane bağlı liste bulundurulacaktır. Bu bağlı listelerin her birinde o mertebedeki boş bloklar tutulacaktır. Örneğin eldeki boş bloklar şöyle olsun:

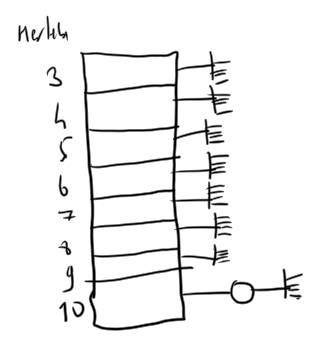


Şüphesiz aynı bağlı listedeki boş bloklar birbirlerinin ikizi değildir (çünkü ikizi olsalardı zaten birleştirilmiş olurlardı). Bu boş blokların oluşturduğu bağlı listeler şöyle gösterilebilir:

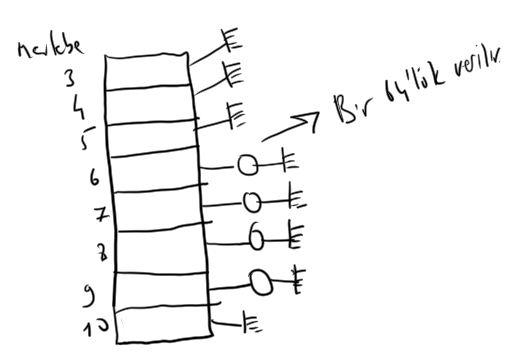


Pekiyi bu bağlı listelerin düğümleri (linkleri) nerededir? Aslında düğümler boş blok listesi tekniğinde olduğu gibi blokların içerisinde, onların başlarında tutulabilir. Blok tahsis edilince zaten bu düğümlere de gerek kalmayacaktır. Pekiyi bu bağlı listeler neden oluşturulmaktadır? İşte tahsisat sırasında önce hangi mertebeden tahsisat yapılacağı belirlenir, sonra da bu bağlı liste dizisinin ilgili elemanının belirttiği bağlı listenin hemen başından eleman alınır. Böylece tahsisat işlemi ek maliyetli sabit zamanlı (amortized constant time) bir işlem olarak yapılabilmektedir.

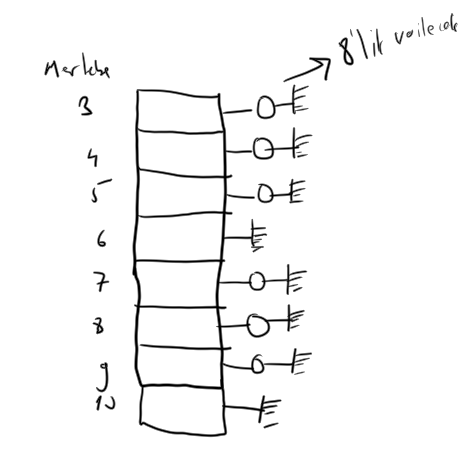
Yukarıda da belirttiğimiz gibi işin başında tipik olarak bu bağlı liste dizisinin içerisinde tek bir eleman vardır:



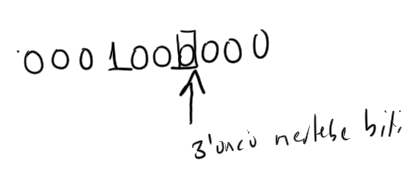
Belli bir mertebeden blok tahsisat istendiğinde bu bağlı liste dizisinde öncelikle o mertebeye ilişkin bağlı listenin boş olup olmadığına bakılır. Eğer o liste boşsa mertebe olarak yukarıya doğru ilk boş olmayan bağlı liste bulunur. Sonra o liste parçalanarak aşağıya doğru istediğimiz mertebeye kadar birer eleman eklene eklene geri gelinir. Örneğin ilk durumda 64’lük bir blok tahsis edilmek istenmiş olsun:



Burada 1024’lük blok bölünerek 64’lük bloğa kadar gelinmiştir. 64’lük bloğun ikizlerinden biri bağlı listeye eklenirken diğeri tahsis etmek isteyen kişiye verilmiştir. (1024 = 512 + 256 + 128 + 64 + 64 olduğuna dikkat ediniz). Artık burada dolu bağlı liste mertebelerinden tahsisat yapılmak istendiğinde hiç blok bölmesi yapılmadan doğrudan o listeden blok verilecektir. Pekiyi yukarıdaki son durumda biz 8’lik bir blok tahsis edilmek istense ne olur? Bu durumda 64’lük blok benzer biçimde bölünerek 8’lik bloklardan biri verilir:



Şimdi bir bloğun ikizinin yerinin belirlenmesi sürecine bakalım. Elimizde bir blok adresi varsa ve o bloğun mertebesini biliyorsak onun ikizinin yerini bulabilir miyiz? Örneğin heap’in 0’dan 1024’e kadar adreslendiğini varsayalım. 64 adresine sahip 3’üncü mertebeden bloğun ikizinin yeri neresidir? Öncelikle bu bloğun ikizi bu bloğun ya solundadır ya da sağındadır. Biz 64 değerini 23 değerine bölerek elde edilen değerin tek mi çift mi olduğuna bakabiliriz. Elde edilen değer çift ise bizim bloğumuzun ikizi onun sağındadır, tek ise solundadır. Örneğin 64 / 8 = 8’dir. 8 çift olduğu için bu bloğun ikizi onun sağındadır. Yani onun 8 byte ilerisindedir. Tabii bu işlemler bit düzeyinde çok daha pratik yapılabilmektedir. Şöyle ki: k adresindeki 2n uzunluğundaki bloğun ikizinin yerini bulmak isteyelim. k değerinin ikilik sistemdeki n’inci bitinin durumu zaten onun ikizinin yeri hakkında bize bilgi vermektedir. Örneğin 64’üncü adresteki 23 = 8 uzunluğundaki bloğun ikizinin yerini bulmaya çalışalım:



İşte eğer mertebeye ilişkin bit 1 ise onun ikizi onu sıfır yaparak (yani 2n kadar geriye giderek), 0 ise onu 1 yaparak (yani 2n kadar ileriye gidilerek) elde edilebilir. EXOR işleminde 0’ın etkisiz eleman olduğunu 1’in ise evrik almakta kullanıldığını anımsayınız. O halde k adresinde ve 2n uzunluğunda bir bloğun ikizinin yeri şöyle bulunabilir::

buddyAddr = (void \*) ((unsigned long) k ^ 1 << n);

Pekiyi ikiz blok sisteminde bir blok serbest bırakılmak istendiğinde neler yapılacaktır? Öncelikle serbest hale getirilecek bloğu ilgili bağlı listeye eklemeden önce onun ikizinin serbest durumda olup olmadığına bakmak gerekir. Eğer bloğun ikizi de serbest durumdaysa ikizini bağlı listeden çıkartıp birleştirerek üst mertebedeki bağlı listeye eklemek gerekir. Fakat eklemeden önce yine o mertebedeki ikizin de serbest olup olmadığına bakılması gerekir. Bu biçimde birleştirme yapılamayana kadar ilerlenir.

Pekiyi serbest bırakılan bloğun ikizinin boş olup olmadığını nasıl anlayabiliriz? Bunu anlamak için ilgili mertebeye ilişkin bağlı listeyi dolaşmak etkin yöntem değildir. (Başarısız aramalarda bağlı listenin sonuna kadar gidileceğine dikkat ediniz.) Bu nedenle programcılar bloğun ikizinin durumu için genellikle ayrı bit dizileri kullanma yoluna giderler. Şöyle ki: Her mertebe için bir bağlı listenin yanı sıra o mertebedeki blokların durumunu gösteren bir bit dizisi de oluşturulur. Örneğin 64 numaralı adresteki 3’üncü mertebeden bloğun ikizinin bit dizisi içerisindeki yeri 64 / 8 = 8 + 1 = 9’dur. Biz bu bite bakarak onun boş olup olmadığını anlayabiliriz. Tabii tahsisat sırasında o bitin de set edilmesi ve serbest bırakma sırasında reset edilmesi gerekmektedir. Aslında buradaki bit dizilerinin ilgili mertebedeki blokların sayısı kadar değil onun yarısı kadar uzunlukta açılması da mümkündür. Örneğin Linux’taki gerçekleştirimde bit dizileri ilgili mertebedeki blok sayılarının yarısı kadardır. Yani her ikiz için tek bit tutulmaktadır. O bit 1 ise ikizlerden biri serbest durumdadır ve birleştirme yapılabilir. Sıfır ise birleştirme yapılamaz. İlgili bit sıfırken ikizlerden biri serbest bırakılmışsa o bit 1 yapılmaktadır. Bazı programcılar bloğun ikizinin boş olup olmadığını bit dizileriyle değil bizzat ikizin içerisinde onun baş kısmında da tutulabilmektedir. Böylece bir blok serbest bırakılırken onun ikizinin yeri belirlenir ve onun baş kısmına bakılarak bloğun boş olup olmadığına karar verilir. Tabii bu durumda bloğun başındaki başlık kısmı tahsis eden tarafından kullanılamayacaktır.

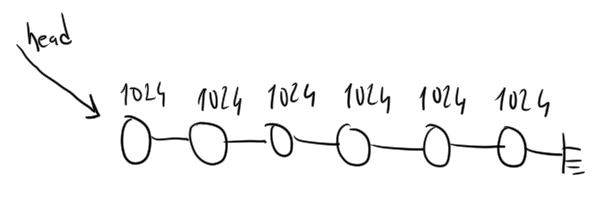
Şimdi başka bir soruna bakalım: İkiz blok sisteminin veri yapısı oluşturulurken bu metadata bilgileri nerede tutulur? Örneğin bağlı listeler ve bit dizileri nerede organize edilmektedir? İşte bu metadata’lar heap olarak belirlenen alanın başında bir yerde oluşturulabilir. Böylece geri kalan alan tahsisat için kullanılabilir. Ya da tamamen başka bir de oluşturulabilir. Yukarıda da belirtildiği gibi bağlı listelerin düğümleri doğrudan bloklar içerisinde, o blokların başında tutulabilmektedir. Fakat blok tahsis edilince o bağların bir önemi kalmadığı için bu düğüm alanı tahsis edilen bloğa dahil edilebilir.

Pekiyi bağlı liste dizilerindeki bağlı listeler tek bağlı mı çift bağlı mı olmalıdır? Serbest bırakma sırasında bloğun ikizinin yerini tepit edip onu bağlı listeden hızlı bir biçimde çıkartmak için listelerin çift bağlı olması daha uygundur. (Çift bağlı listelerde adresi bilinen bir düğümün sabit zamanlı olarak silinebildiğini anımsayınız.)

**2.3.3. Dilimli Tahsisat Sistemi (Slab Allocator)**

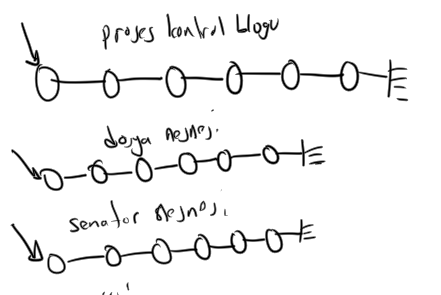
Dilimli tatsisat sistemi özellikle işletim sistemlerinin çekirdek heap sistemlerinin gerçekleştirilmesinde yaygın olarak kullanılmaktadır. Örneğin Solaris, Linux ve BSD sistemleri kendi çekirdekleri içerisindeki tahsisatlarda dilimli tahsisat sistemini kullanmaktadır. Dilimli tahsisat sistemi ilk kez Jeff Bonwick tarafından Sun OS 5.4 sistemlerinde uygulanmıştır. (Bonwick tarafından yazılan “The Slab Allocator: An Object-Caching Kernel Memory Allocator” makalesini inceleyiniz).

Dilimli tahsisat sisteminin dayandığı fikir basittir: Boş blokların bağlı listede tutulması tekniğinde eğer tüm boş bloklar aynı uzunlukta olsaydı tahsisat ve geri bırakma işlemleri çok hızlı yapılırdı değil mi? Çünkü bu durumda bu boş bağlı listede arama yapmaya gerek kalmazdı. Örneğin tüm boş blokların 1024 byte uzunluğunda olduğunu varsayalım. Şimdi biz 1024 byte tahsis etmek istediğimizde sistem hemen bağlı listenin önündeki bloğu bize verebilir:

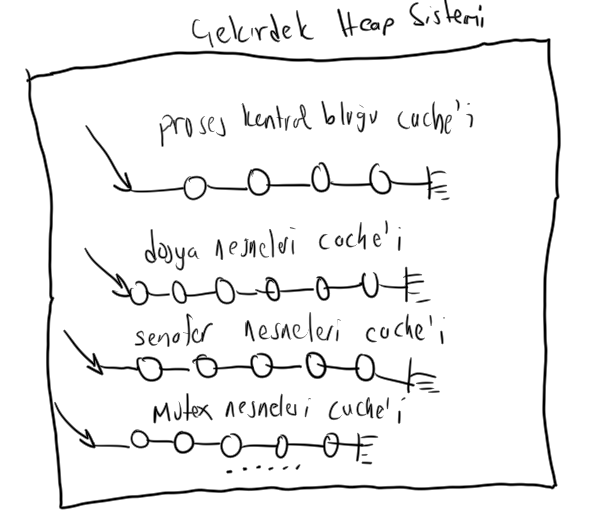


Tabii böyle bir sistemde tahsis edilecek blokların hep aynı uzunlukta olacağı varsayılmaktadır. Bu varsayım altında bir bloğun serbest bırakılması durumunda bir birleştirme yapmanın da gereksiz olduğu aşikardır.

Pekiyi hep aynı uzunlukta tahsisat yapma gerçekçi bir durum mudur? Daha büyük ya da daha küçük blok tahsis etmek istersek ne olacaktır? İşte dilimli tahsisat sistemi genel amaçlı bir sistem olmaktan çok özel amaçlı bir sistemdir. Örneğin işletim sistemlerinin çekirdeklerinde dinamik biçimde tahsis edilecek pek çok yapı vardır (proses kontrol blokları, dosya nesneleri, senkronizasyon nesneleri vs.) İşte uzunluğu zaten baştan belli olan bu nesneler için ayrı ayrı (her biri ayrı uzunlukta) dilimli tahsisat sistemleri oluşturulabilir. Böylece aşağıdaki gibi farklı uzunlukta nesneleri tutan ayrı boş blok listeleri elde edilmiş olacaktır:



Dilimli tahsisat sisteminde farklı uzunluklar için oluşturulmuş her tahsisat sistemlerine “cache” denilmektedir. Bu sistemde önce belli bir uzunluk belirtilerek bir “cache” oluşturulur. Sonra tahsis etme ve geri bırakma işlemleri cache belirtilerek belli bir cache'ten yapılır. Böylece örneğin çekirdek yazılımcısı bir tane “proses kontrol bloğu” tahsis edecekse onu “proses kontrol bloğu cache’inden” tahsis eder.



Linux işletim sisteminde yeni bir cache yaratmak için kmem\_cache\_create isimli çekirdek fonksiyonu kullanılmaktadır. Örneğin Linux 2.4 çekirdeğindeki kmem\_cache\_create fonksiyonunun prototipi şöyledir:

[kmem\_cache\_t](http://fxr.watson.org/fxr/ident?v=linux-2.4.22;i=kmem_cache_t) \*[kmem\_cache\_create](http://fxr.watson.org/fxr/ident?v=linux-2.4.22;i=kmem_cache_create) (const char \*[name](http://fxr.watson.org/fxr/ident?v=linux-2.4.22;i=name), [size\_t](http://fxr.watson.org/fxr/ident?v=linux-2.4.22;i=size_t) [size](http://fxr.watson.org/fxr/ident?v=linux-2.4.22;i=size), [size\_t](http://fxr.watson.org/fxr/ident?v=linux-2.4.22;i=size_t) [offset](http://fxr.watson.org/fxr/ident?v=linux-2.4.22;i=offset),

unsigned long [flags](http://fxr.watson.org/fxr/ident?v=linux-2.4.22;i=flags), void (\*ctor)(void\*, [kmem\_cache\_t](http://fxr.watson.org/fxr/ident?v=linux-2.4.22;i=kmem_cache_t) \*, unsigned long),

void (\*dtor)(void\*, [kmem\_cache\_t](http://fxr.watson.org/fxr/ident?v=linux-2.4.22;i=kmem_cache_t) \*, unsigned long));

Bu fonksiyon bir cache yaratır. Cache bilgileri kmem\_cache\_t isimli bir yapı ile temsil edilmektedir. Fonksiyon yaratılan cache'e ilişkin cache bilgilerinin tutulduğu yapının adresini bize bir handle değeri gibi vermektedir. (Bu handle değerine Linux terminoloisinde “cache descriptor” deniyor.) Linux’ta dilimli tahsisat sisteminde kullanılan cache’lerin handle alanları da (yani kmeme\_cache\_t yapıları da) yine dilimli tahsisat sistemi ile tahsis edilmektedir. Dilimli tahsisat sisteminin kendi veri yapıları için kullanılan cache “cache\_cache” biçiminde isimlendirilmektedir. Yaratılan bir cache kmem\_cache\_destroy fonksiyonuyla serbest bırakılır.

Linux’ta belli bir cache’ten (yani belli uzunluk için yaratılmış cache’ten) tahsisat yapmak için kmem\_cache\_alloc fonksiyonu kullanılmaktadır:

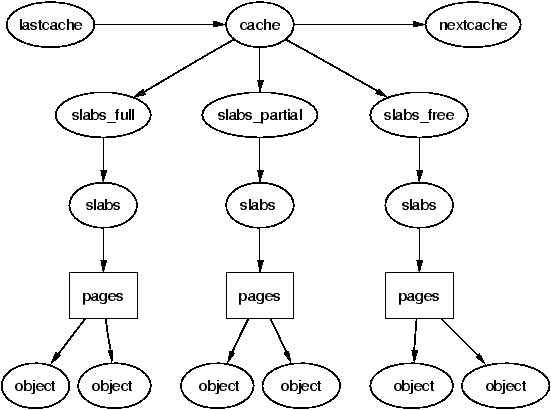
void \*[kmem\_cache\_alloc](http://fxr.watson.org/fxr/ident?v=linux-2.4.22;i=kmem_cache_alloc)([kmem\_cache\_t](http://fxr.watson.org/fxr/ident?v=linux-2.4.22;i=kmem_cache_t) \*cachep, int [flags](http://fxr.watson.org/fxr/ident?v=linux-2.4.22;i=flags));

kmem\_cache\_alloc fonksiyonuyla tahis edilen blok kmem\_cache\_free fonksiyonuyla serbest bırakılmaktadır:

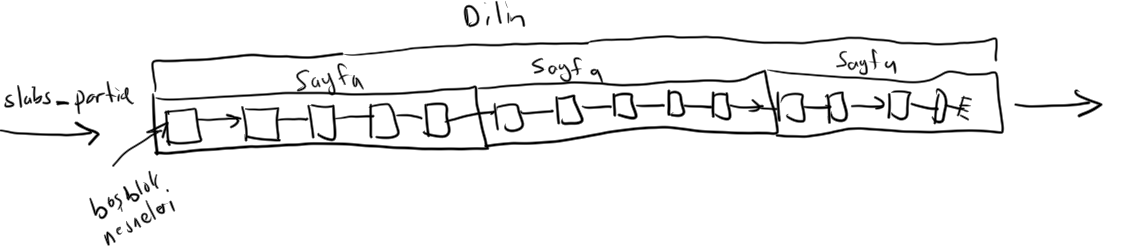
void [kmem\_cache\_free](http://fxr.watson.org/fxr/ident?v=linux-2.4.22;i=kmem_cache_free)([kmem\_cache\_t](http://fxr.watson.org/fxr/ident?v=linux-2.4.22;i=kmem_cache_t) \*cachep, void \*objp);

**2.3.3.1. Dilimli Tahsisat Sisteminin Ayrıntıları**

Dilimli tahsisat sistemi yukarıda da belirtildiği gibi özellikle işletim sistemlerinin çekirdek heap alanlarının organizasyonunda tercih edilmektedir. Öncelikle şu soruya yanıt arayalım: Dilimli tahsisat sistemindeki “dilim (slab)” terimi ne anlam ifade etmektedir? Bir dilim (slab) ardışıl n tane sayfadan (page) oluşan bir heap bloğudur. İşletim sistemi düzeyinde en aşağı seviyeli olarak tahsisatlar sayfa temelinde yapılmaktadır. Yukarıda da belirttiğimiz gibi sayfa tahsisatları için genellikle ikiz blok sistemi (buddy allocator) tercih edilmektedir. İşte dilimli tahsisat sistemi için bir cache oluşturulduğunda o cache de dilimlerden oluşmaktadır. Tahsis edilecek bloklar dilimlerin içerisindedir ve onlara bu terminolojide nesne (object) denir. Cache sistemi önce bir dilimle başlatılıt, sonra yetmezse başka dilimler de cache’e dahil edilir. Bir dilimdeki nesnelerin hepsi serbest bırakıldığında o dilim hemen çekirdeğin aşağı seviyeli sayfa tahsisat sistemiyle (ikiz blok sistemine) hemen iade edilmez. Daha sonra gereksinim duyulabilir diye bekletilir. Dilimli tahsisat sistemi için bir cache sistemini aşağıdaki gibi bir şekille temsil edebiliriz:



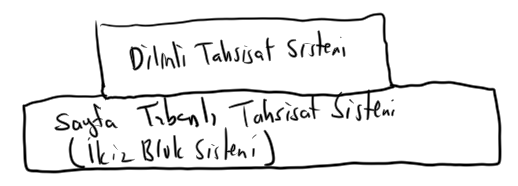
Bu şekil Linux işletim sisteminin çekirdeğini anlatan bir dokümandan alınmıştır. Linux çekirdeği bir dilimli tahsisat sistemi için bir cache yarattığında o cache içerisinde 3 tür dilimi bağlı listeler halinde tutmaktadır: Tam boş olan dilimler (slabs\_free), tam dolu olan dilimler (slabs\_full) ve tam dolu olmayan (ya da tam boş olmayan) dilimler (slabs\_partial). Cache içerisindeki boş blok listeleri dilimlerin dolayısıyla sayfaların içerisindedir. Örneğin:



Dilimli tahsisat sistemindeki sayfalar, dilimler ve nesneler ile ilgili önemli sorular ve yanıtları şöyledir:

**Soru:** Dilimli tahsisat sistemi çekirdeğin en aşağı seviyeli tahsisat sistemi midir?

**Yanıt:** Hayır. Dilimli tahsisat sistemi işletim sistemlerinde sayfa tabanlı tahsisat işlemlerini yapan tahsisat sistemini (tipik olarak ikiz blok sistemini) kullanan daha yüksek seviyeli bir tahsisat sistemidir.



Yani dilimli tahsisat sistemi tahsisatta kullanacağı dilimleri (yani heap olarak kullanacağı alanı) sayfa düzeyinde tahsisat yapan alt sistemden istemektedir.

**Soru:** Dilimli tahsisat sisteminde cache'te neden birden fazla dilim (slab) vardır örneğin neden büyük ve tek bir dilim yoktur?

**Yanıt:** Cache’in dilimlerden oluşmasının nedeni belli dilimler tamamen boşaltıldığında sayfa tabanlı tahsisat sistemine onların geri verilmesini sağlamaktır. Eğer cache’te yalnızca bir tane büyük bir dilim bulunsaydı o büyük dilim sürekli bir kısmı iade edilemeden bekletilirdi.

**Soru:** Bir dilim neden ardışıl n tane sayfadan oluşmaktadır da bir tane sayfadan oluşmamaktadır?

**Yanıt:** Şüphesiz bu durum mümkün olabildi. Ancak bir dilimin bir sayafadan oluşması bazı bakımlardan etkin değildir. Çünkü bir sayfa nispeten küçük bir birimdir (Intel’de 4K). Eğer bir dilim 1 sayfadan oluşturulsa bu kez cache içerisinde pek çok dilim birikir ki, bunların da bağlı liste içeirisnde tutulması ve işlenmesi zor olur. Ayrıca dilimlerin tek sayfadan oluşması durumunda tahsisat sistemi için gereken toplam "metadata" alanları da oransal olarak büyüyecektir.

**Soru:** Dilimli tahsisat sisteminde gerçek tahsisatı yapan fonksiyonlar (örneğin Linux’taki kmem\_cache\_alloc) tahsisatı hangi dilimlerden yapmaktadır?

**Yanıt:** Cache’teki hepsi dolu olmayan (slabs\_partial) dilimleri içerisindeki dilimlerden.

**Soru:** Bir cache’te aynı uzunluktaki nesneleri tutan bağlı listelerden kaç tane vardır?

**Yanıt:** Hepsi dolu olmayan her dilim için bu listeden bir tane vardır. Ayrıca Linux sistemlerindeki dilimli tahsisat sistemi gerçekleştiriminde boş blokların kendileri değil onların indeks numaraları bağlı listede tutulmaktadır (Bunun için çekirdek kodlarını inceleyiniz. Çekirdek kodları içerisindeki slab\_t yapısının kmem\_bufctl\_t elemanı aslında int türden indekslerden oluşan bir bağlı liste dizisidir.)

**Soru:** Cache içerisindetamamen boşaltılmış dilimler ne zaman sayfa tabanlı tahsisat sistemine iade edilmektedir?

**Yanıt:** Boşaltılmış dilimler cache’te daha sonra gereksnimin duyulur diye bekletilirler. Zaten buna “cache” denmesinin bir nedeni de budur. Çekirdek başka amaçlarla sayfalara gereksinim duyduğunda bu boş sayfalar iade edilmektedir.

**Soru:** Belli bir anda tüm dilimler doluyken tahsisat yapılmak istenirse ne olur?

**Yanıt:** Bu durumda sayfa tabanlı tahsisat sisteminden yeni bir dilim tahsis edilerek cache'e dahil edilir.

Dilimli tahsisat sisteminin önemli bir özelliği de tahsis edilen nesneler için yapılan başlangıç ve bitiş işlemlerinin (bunları nesne yönelimli teknikteki "constructor" ve "destructor"lara benzetebilirsiniz) mümkün olduğunca az yapılmasını sağlamasıdır. Şöyle ki: Bu sistemde bir nesne tahsis edildiğinde o nesnenin elemanlarını için yapılan birtakım ilkdeğer verme işlemleri ve diğer işlemler (örneğin nesnenin bir elemanı "semaphore" olabilir, işin başında bu "semaphore"un yaratılması gerekir) yalnızca bir kez yapılmaktadır. Yani nesne ilk kez tahsis edildiğinde bu ilk işlemler bir kez yapılır. Sonra nesne boşaltıldığında bitiş işlemleri uygulanmadan nesne cache’te bekletilir. Daha sonraki bir tahsisatta sistem bu nesneyi yeniden verdiğinde bu ilk işlemler gereksiz biçimde yeniden yapılmayacaktır. Benzer biçimde nesnenin yok edilmesi sırasında yapılacak son işlemler de nesne serbest bırakılırken değil, tüm dilim sayfa tabanlı tahsisat sistemine iade edilirken yalnızca bir kez yapılmaktadır. Daha önce vermiş olduğumuz kmem\_cache\_create fonksiyonun prototipine bir kez daha bakınız:

[kmem\_cache\_t](http://fxr.watson.org/fxr/ident?v=linux-2.4.22;i=kmem_cache_t) \*[kmem\_cache\_create](http://fxr.watson.org/fxr/ident?v=linux-2.4.22;i=kmem_cache_create) (const char \*[name](http://fxr.watson.org/fxr/ident?v=linux-2.4.22;i=name), [size\_t](http://fxr.watson.org/fxr/ident?v=linux-2.4.22;i=size_t) [size](http://fxr.watson.org/fxr/ident?v=linux-2.4.22;i=size), [size\_t](http://fxr.watson.org/fxr/ident?v=linux-2.4.22;i=size_t) [offset](http://fxr.watson.org/fxr/ident?v=linux-2.4.22;i=offset),

unsigned long [flags](http://fxr.watson.org/fxr/ident?v=linux-2.4.22;i=flags), void (\*ctor)(void\*, [kmem\_cache\_t](http://fxr.watson.org/fxr/ident?v=linux-2.4.22;i=kmem_cache_t) \*, unsigned long),

void (\*dtor)(void\*, [kmem\_cache\_t](http://fxr.watson.org/fxr/ident?v=linux-2.4.22;i=kmem_cache_t) \*, unsigned long));

Bu fonksiyondaki ctor fonksiyon göstericisi bir nesne tahsis edildiğinde o nesneyi ilklemek (initialize etmek) için yalnızca bir kez çağrılacak fonksiyonu, dtor fonksiyon göstericisi de dilim sayfa tabanlı tahsisat sistemine iade edildiğinde bitiş işlemleri için yalnızca bir kez çağrılacak fonksiyonu belirtmektedir.

Linux işletim sisteminde (dilim tahsisat sistemini kullanan diğer sistemlerde de böyle) belli veri yapıları için oluşturulmuş cache’lerin dışında bir de 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536 ve 131072 uzunlukta genel amaçlı cache’ler bulundurmaktadır. Bu cache’ler sistem açılırken yaratılırlar. Böylece örneğin bir aygıt sürücüsü yazan programcı kendi amaçları için çekirdeğin heap’inden n byte tahsis etmek isterse bu n değerinden büyük olan en küçük cache’ten tahsisatını yapar. Aslında Linux’ta bu işlemi tek hamlede yapan kmalloc isimli bir çekirdek fonksiyonu bulunmaktadır:

void \*[kmalloc](http://fxr.watson.org/fxr/ident?v=linux-2.4.22;i=kmalloc) ([size\_t](http://fxr.watson.org/fxr/ident?v=linux-2.4.22;i=size_t) [size](http://fxr.watson.org/fxr/ident?v=linux-2.4.22;i=size), int [flags](http://fxr.watson.org/fxr/ident?v=linux-2.4.22;i=flags));

Fonksiyonun “cache descriptor” istemediğine dikkat ediniz. Fonksiyon doğrudan birinci parametresiyle belirtilen uzunluktan büyük olan en küçük cache’i hesaplar ve tahsisatını o cache’ten yapar. kmalloc ile tahsis edilen blok kfree fonksiyonuyla geri bırakılabilmektedir:

void [kfree](http://fxr.watson.org/fxr/ident?v=linux-2.4.22;i=kfree) (const void \*objp);

**2.3.3.2. Dilimli Tahsisat Sisteminin Performans Kıyaslaması**

Bu sistemde belli bir cache’ten tahsisat yapmak ve onu serbest bırakmak sabit karmaşıklıkta yani çok hızlıdır. Zaten bu sistemin işletim sistemlerinin çekirdekleri tarafından tercih edilmesinin en önemli nedeni hızlı oluşudur. Dilimli tahsisat sisteminde bir cache’ten yapılan tahsisatlarda içsel bölünme sıfır düzeyindedir. Tüm nesneler aynı uzunlukta olduğu için dışsal bölünmenin de sıfır düzeyinde olduğu söylenebilir. Ancak dilimli tahsisat sistemi uzunluğu baştan bilinen veri yapıları için düşünülmüştür. Dolayısıyla bu sistemde her veri yapısı için ayrı cache’in oluşturulması gerekir. Bu nedenle bu sistem “user mode” programlar için genel ve etkin bir tahsisat sistemi olamaz. (Anımsanacağı gibi genel tahsisat fonksiyonları klasik boş blok bağlı liste tekniğini kullanmakatdır.) Ayrıca belki belirtmemize gerek yok fakat dilimli tahsisat sisteminin gerçekleştirimi biraz zordur.

**3. Derleyicilerin ve Yorumlayıcıların Tasarımları ve Gerçekleştirimleri**

Bu bölümde derleyicilerin ve yorumlayıcıların tasarım ve gerçekleştirim prensipleri ele alınacak ve açıklanacaktır. Bu bölümün sonunda kurs katılımcılarının çeşitli araçları kullanarak basit yorumlayıcılar yazabilmesi öngörülmektedir.

**3.1. Dil Olgusu ve Dillerin Sınıflandırılması**

Dil karmaşık bir olgudur. Pek çok bilimin çalışma alanı içerisine girmektedir. (Örneğin dilbilim (linguistics), bilişsel bilimler (cognitive science), psikoloji (psychology), sosyoloji (sociolgy) vs.) Bu nedenle dilin basit bir tanımını yapmak zordur. Pek çok teorisyen ve düşünür değişik tanımlar yapmışlardır. Ancak bu tanımların hepsinde ortak özelliklerden biri dilin “iletişimde kullanılan bir araç” olduğudur. Bir dilin pek çok kural topluluğu söz konusu olabiliyorsa da en temel iki kural kümesinden bahsedilebilir: Sentaks ve semantik. Sentaks ve semantik dili dil yapan ve tüm dillerde var olan ortak özelliklerdir.

Sentaks dili oluşturan en yalın öğelerin (bunlara atom (token) denir) doğru yazılma ve dizilme kurallarıdır. Örneğin aşağıdaki İngilizce cümlede öğeler doğru dizilmemiştir:

I going am school to

Bu İngilizce’ye göre bir sentaks hatasıdır. Aşağıdaki cümlede de Türkçe’ye göre bir sentaks hatası vardır:

Herkez çok neşeliydi

"Herkez" sözcüğü yanlış yazılmıştır. Sentaks yalnızca doğal dillerde değil programlama dillerinde de söz konusu olan bir kurallar kümesidir. Örneğin:

if )a == 10)

printf(“ok\n”);

Burada if anahtar sözcüğünden sonra ‘(‘ atomunun gelmesi gerekirdi. Halbuki ‘)‘ atomu gelmiştir. Bu da bir sentaks hatasıdır.

Semantik doğru yazılmış ve dizilmiş öğelerin ne anlam ifade ettiğine ilişkin kurallardır. Yani örneğin “I am going to school” doğru yazılmış ve dizilmiştir. Fakat ne anlam ifade etmektedir? Ya da örneğin:

if (a == 10)

printf(“Ok\n”);

Doğru yazılmış ve dizilmiştir fakat ne anlam ifade etmektedir?

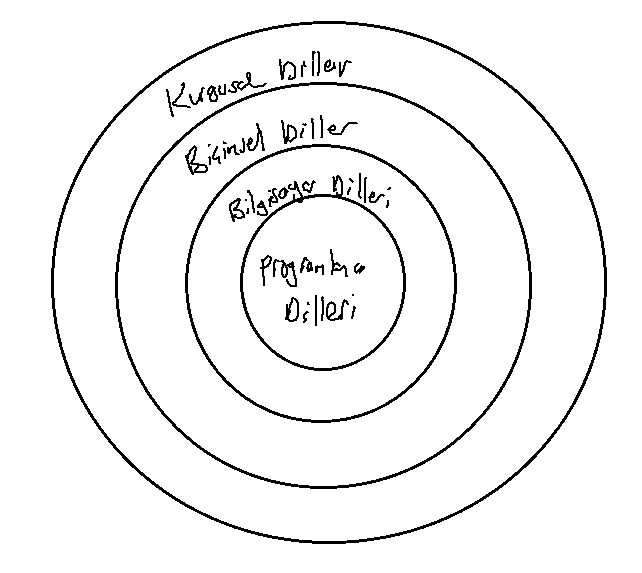
Bir olgunun dil olarak nitelendirilmesi için onun en azından sentaks ve semantik kurallara sahip olması gerekir. Sentaks ve semantik kuralların dışında bazı diller başka kural topluluklarına da sahip olabilirler. Örneğin doğal dillerde “fonetik” teleaffuza ilişkin kurallarla ilgilidir.

**3.1.1. Doğal Diller, Kurgusal ve Biçimsel Diller**

Yaşantı sonucuyla oluşmuş dillere doğal diller denir. Doğal diller son derece karmaşık sentaks ve semantik yapıya sahiptir. Doğal dillerin sentaks kuralları tam olarak matematiksel biçimde ifade edilememektedir. İnsanların belli bir amaç için tasarladığı dillere ise kurgusal diller (constructed languages) denilmektedir. Sentaksları tam olarak matematiksel biçimde ifade edilebilen diller ise ise biçimsel diller olarak isimlendirilir. Biçimsel dillerde doğal dillerdeki gibi istisnalar yoktur. Çünkü istisnalar aslında sentaks kurallarını bozucu bir etki yapmaktadır. Doğal dilleri öğrenirken kuralları olduğunu sandığımız pek çok yapının aslında çok fazla istisnalarının olduğunu görmüşüzdür. İstisnalar ise öğrenmeyi zorlaştırmaktadır.

**3.1.2. Bilisayar Dilleri ve Programlama Dilleri**

Bilgisayar sistemlerinde kullanılmak üzere tasarlanmış dillere bilgisayar dilleri 8computer languages) denilmektedir. Yani bir olgunun “bilgisayar dili” olarak nitelendirilmesi için onun bilgisayar dünyası için tasarlanmış olması ve sentaks, semantik kurallara sahip olması gerekir. Bu bakımdan örneğin XML bir bilgisayar dilidir. (XML "Extensible Markup Language" sözcüklerinden kısaltılmıştır.) Bu dilde sentaks ve semantik kurallar vardır. Bu dil bilgisayar sistemleri tarafından kullanılmak üzere tasarlanmıştır. Bilgisayar dillerinde bir “akış (flow)” olması zorunlu değildir. Eğer bir bilgisayar dilinde bir akış da varsa bu tür dillere “programlama dilleri (programming languages)” denilmektedir. Örneğin C bir programlama dilidir, ancak XML bir programalam dili değildir.



Pekiyi örneğin UML (Unified Modeling Language) bir dil midir? İsminden de anlaşılabileceği gibi UML bir dildir. UML’in özellikle nesne yönelimli projeleri modellemek için kullanılan diyagramlardan oluşmuş bir yapısı vardır. Bu diagramları çizmenin bir kuralı vardır. Bu kural UML’in sentaksını oluşturur. Tabii bu diagramların anlamları da vardır. Bu da dilin semantiğini oluşturmaktadır. UML bazılarına göre bir bilgisayar dili olarak nitelendirilebilir bazılarına göre ise kurgusal bir dil olarak sınıflandırılmaktadır (çünkü UML bilgisayarla hiçbir ilgisi olmayan endüstri alanlarında da kullanılmaktadır).

**3.1.3. Biçimsel Dillerin Teorik Altyapısı**

Eskiden diller matematiksel bir modelle ele alınmıyordu. Biçimsel diller (formal languages) dillerin matematiksel bir bakış açısı ile ele alınması süreci sırasında ortaya çıkmıştır. Bazı çalışmalar çok daha eskiye dayanıyorsa da bu konudaki modern altyapı büyük ölçüde Noam Chomsky’nin çalışmalarıyla oluşturulmuştur. Chomsky dillerin sentakslarını matematiksel terimlerle açıklamış ve “üretici gramer (generative grammar)” kavramını ortaya atmıştır. Gerçekten bilgisayar dillerinin resmi sentakslarını açıklayan BNF ve EBNF gibi notasyonlar Chomsky’nin bu çalışmalarından ilham alınarak geliştirilmişlerdir.

Chomsky dilleri sentaks üretim biçimlerine göre dört bölüme ayırmıştır:

**Type 0:** Bu dillerin sentaksları oldukça karmaşıktır. Doğal diller bu tür sentaks yapıları içerirler. Bu dillere “serbest (free)” sentakslar da denilmektedir.

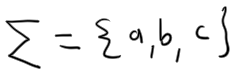
**Type 1:** Bu sentaks biçimine sahip dillere “bağlam bağımlı (context sensitive)” diller de denilmektedir. Bu tür gramerlerde bir öğenin açılımı (ve anlamı) hangi bağlamda bulunduğuna bağlı olarak değişebilmektedir.

**Type 2:** Bu sentaks yapısına sahip olan dillere de “bağlam bağımsız (context free)” diller de denilmektedir. Bağlam bağımsız dillerin sentakslarında bir öğe hangi bağlamda olursa olsun hep aynı biçimde açılmaktadır (yani hep aynı anlama gelmektedir). Modern programlama dillerinin pek çoğu bağlam bağımsız bir sentaksla ifade edilebilmektedir. Bu nedenle bu dillere “bağlam bağımsız diller (context free langueages)” de denilmektedir.

**Type 3:** Bu tür sentakslara Chomsky “düzenli gramer (regular grammar)” demektedir. Pek çok kütüphanede kullanılan düzenli ifadeler (regular expressions) düzenli gramere örnek verilebilir.

**3.1.4. Biçimsel Dillerin Matematiksel İfadeleri**

Diller matematiksel olarak kümeler teorisi kullanılarak ifade edilebilmektedir. Bir dil aslında bir semboller kümesidir. Bir dili oluşturan en yalın elemanlara “son semboller (terminal symbols)” ya da "alfabe (alphabet)" denir. Alfabeyi oluşturan küme sigma işareti ile gösterilmektedir. Örneğin:



Alfabedeki karakterlerin art arda getirilmesiyle oluşan dizilimlere "string" denilmektedir. Örneğin yukarıdaki sonlu sembol kümesindeki bazı dizilimler şunlar olabilir:

aabb

abcca

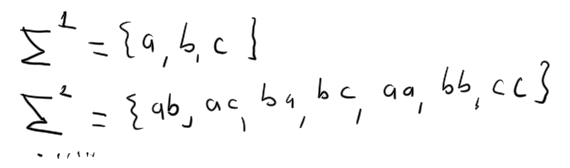
abccab

aaaaaaaaaa

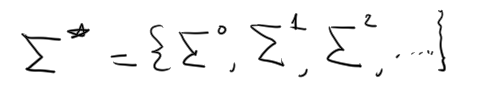
Sonlu semboller kümesi sayılabilir sonlu bir kümedir. Ancak bundan elde edilebilecek tüm string’lerin kümesi sonlu bir küme değildir.

Bir alfabede her zaman boş kümeye karşı gelen bir elemanın bulunduğu kabul edilir. Bu eleman λ ile gösterilmektedir.

Bir alfabedeki n elemanlı string’lerin kümesi sigma karalterinin sağ köşesine n sayısı yazılarak gösterilebilir. Örneğin:



Sigma \* tüm bu kümelerin birleşimini ifade eder:



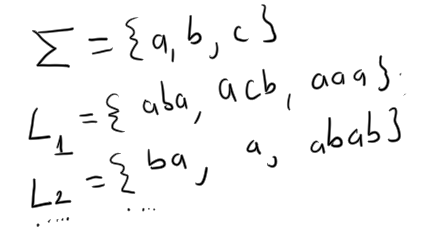
Örneğin:

Sigmanın a ve b elemanlarından oluştuğunu kabul edelim. Bu durumda Sigma \* aşağıdaki gibi bir küme olacaktır:



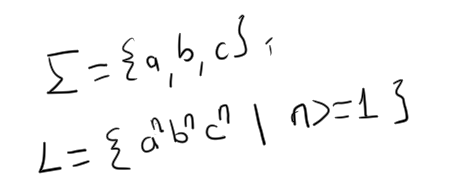
Başka bir deyişle sigma \* alfabeden teorik olarak elde edilecek tüm stringlerin kümesidir. Yukarıda da belirttiğimiz gibi sigma \* kümesinin eleman sayısı sonsuzdur.

İşte sigma \* kümesinin herhangi bir alt kümesine "dil (language)" denilmektedir. Örneğin:



O halde bir dili tanımlayabilmek için öncelikle bir alfabenin tanımlanmış olması gerekir. Örneğin Türkçe için alfabe 29 harften oluşmaktadır. C Programlala Dilinin alfabesi ise pek çok alfa numerik karakterlerden oluşmaktadır.

Bilindiği gibi matematikte bir kümenin eleman sayısı çok fazla ise artık listeleme tekniği gösterim için elverişsiz olmaktadır. Onun yerine ortak özellik (öyle ki) tekniği kullanılır. Örneğin:



Buradaki L dilinin bazı elemanları şunlardır: abc, aabbcc, aaabbbccc, ... Görüldüğü gibi buradaki L dilinin elemanları sonsuz sayıdadır. Sonlu sayıda elemandan oluşan alfabeden sonsuz sayıda elemana sahip dil elde edilebildiğine dikkat ediniz.

Ancak bir dilin eleman sayısı çok fazlaysa ya da sonsuz sayıdaysa onu her zaman ortak özellik yöntemine göre ifade edemeyiz. İşte dillerin elemanlarını belirlemek için Chomsky “üretici gramer (generative grammar)” kavramını ortaya atmıştır.

Şimdi C Programlama Dilini biçimsel bir dil olarak ele almaya çalışalım. C’nin alfabesi temel alfanümerik karakterlerden oluşmaktadır. Bu karakterlerin tüm yan yana getirilmiş hallerinin kümesi sigma \*’dır. C'de sigma \* kümesinin bir alt kümesidir. C geçerli olan tüm C programlarının (stringlerinin) oluşturduğu bir kümedir. Geçerli olan her C programı C dilini oluşturan kümedeki bir string'tir. Tabii yukarıda da belirttiimiz gibi C gibi karmaşık bir dilin tüm elemanlarını ortak özellik yöntemine göre yazmak mümkün değildir. İşte C gibi bir dilin tüm elemanlarını üretici gramer tekniği ile ifade edebiliriz.

**3.1.5. Üretici Gramer (Generative Grammar) Kavramı**

Üretici gramer bir dilin tüm elemanlarını üreten bir otomat (automata) olarak düşünülebilir. Üretici gramerin dört öğresi vardır:

G = {V, T, S, P}

Buradaki harflerin kısaltması şöyledir:

V: Variable (genelikle büyük harflerle gösterilmektedir).Ggrameri ifade ederken kullanılan ara sembolleri belirtmektedir.

T: Terminal Symbol (ya da alfabe). T yerine sigma da kullanılabilir. (Kursumuzda "terminal symbol" yerine Türkçe "son sembol" terimini de kullanacağız.)

S: Start Symbol (variable kümesi içerisindeki bir sembol olmak zorundadır.)

P: Production

Gramer başlangıç sembolünden (start symbol) başlar. Her aşamadan bir üretimle (production) geçilir. Üretimler değişken (variable) denilen ara sembolleri kullanır. Bir üretim bir grup sembolün yerine başka bir sembolün yerleştirilmesi anlamına gelmektedir. Bu biçimde üretim devam ettirilir. Ta ki her şey alfabedeki elemanlardan (yani "terminal symbol"lardan) oluşana kadar. Üretimler bir okla gösterilmektedir. Tipik olarak Ok işaretinin solunda bir değişken, sağında da onun nasıl açılacağı (yani onun yerine ne yerleştiriliceği) bilgisi vardır. Aslında ok işaretinin solunda tek bir değişken bulunmak zorunda değildir. Bağlam bağımlı ve serbest dillerin gramerlerinde ok işaretinin solunda birden fazla ara sembol ve son sembol de bulunabilemktedir.

Şimdi üretici bir gramer örneği verelim:

G = {{X}, {a, b}, {X}, P}

P üretimleri şunlardır:

X -> aXb

X -> λ

Bu üretici gramerden ne anlamalıyız? Burada başlangıç sembolü X’tir. Dilin alfabesi (yani sigma kümesi) ise {a, b}’den oluşmaktadır. Değişkenler kümesi de yalnızca X’ten oluşmaktadır. Şimdi biz ok sembolünün solundaki X yerine onun sağındaki dizilimi yerleştirerek bu işleme yalnızca alfabedeki semboller kalana kadar devam edersek dilin bir elemanını buluruz. Bu işlemi her yinelediğimizde de dilin başka bir elemanı elde ederiz. Bu üretici gramer kümesel yöntem yetersiz kaldığı için Chomsky tarafından düşünülmüştür. Örneğin yukarıdaki G dilinin bir elemanı şöyle elde edilebilir:

1) X -> aXb (aXb)

2) X -> λ (ab)

Buradan elde edilen string “ab” dir. Şimdi başka bir elemanı elde edelim:

1) X -> aXb (aXb)

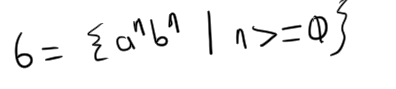
2) X -> aXb (aaXbb)

3 X -> aXb (aaaXbbb)

4) X -> λ (aaabbb)

Buradan elde edilen string de “aaabbb” olacaktır. Burada hiçbir sembol içermeyen bir string’in de (bu λ ile gösteriliyor) bu dile dahil olduğuna dikkat ediniz.

Yukarıdaki üretici gramerle açıklanmış olan basit dil aslında -basit olduğu için- ortak özellik yöntemiyle de gösterilebilirdi:



Biçimsel diller ve üretici gramerler konusuyla ilgili tipik soru ve yanıtlar şunlardır:

**Soru:** Bir dilin alfabesi ne demektir?

**Yanıt:** Dildeki en yalın elemanların oluşturduğu kümeye “alfabe (alphabet)” ya da “son semboller (terminal symbols)” kümesi denir ve büyük harf sigma karakteri ile gösterilir.

**Soru:** Biçimsel dil terminolojisindeki string nedir?

**Yanıt:** Alfabedeki sembollerin peşi sıra getirilmesiyle elde edilen her dizilime string denir. Bir dil aslında string'lerden oluşan bir kümedir.

**Soru:** ∑\* (Sİgma yıldız) ne anlama gelmektedir?

**Yanıt:** ∑\* alfabedeki karakterlerin istenildiği kadar birbirleriyle birleştirilmesiyle oluşturulan tüm string’lerin kümesidir. Dil (language) bu kümenin bir alt kümesi olarak tanımlanmaktadır.

**Soru:** Dil bir string kümesi ise onu nasıl ifade edebiliriz?

**Yanıt:** Eğer bu kümenin eleman sayısı az ise onu doğrudan listeleme yöntemiyle ifade edebiliriz. Örneğin: L = {ababa, aaaaa, bbab} gibi. Ancak eleman sayısı fazla (hatta sonsuz) olan dillerin listeleme yöntemiyle ifade edilmesi olanaksızdır. Ortak özellik yöntemi ise ancak kuralları çok belirgin olan bazı dilleri ifade edebilmektedir. Örneğin: L = {anbn | n >= 1}. İşte bu iki yöntem yetersiz olduğundan dolayı “üretici gramer (generative grammar)” yöntemi geliştirilmiştir. Bu yöntemde soldaki sembol sağdaki ile açılarak ilerlenir ta ki tüm semboller alfabedeki sembollerden oluşuna kadar.

**Soru:** Üretici Gramerin resmi ifadesi nasıldır?

**Yanıt:** Üretici gramerin resmi gösterimi G = {V, T, S, P} biçimindedir. Burada V açılacak başlangıç sembolünü belirtir. T ise alfabeyi belirtmektedir. S açılımın başlatılacağı semboldür. P de açılımda kullanılacak üretimlerin kümesidir. Başlangıç sembolünden başlanarak P üretimlerimleri yoluyla elde edilen açılımlardaki tüm semboller yalnızca alfabedeki semboller olana kadar işlem devam ettirilir.

**Soru:** Üretici gramer bir dildeki stringlerin kümesini nasıl oluşturmaktadır?

**Yanıt:** Başlangıç sembolünden hareketle açılım yapıla yapıla elde edilen tüm stringler dili oluşturmaktadır. Benzer biçimde elde edilemeyen stringler de o dilin bir elemanı değildir.

**Soru:** Bir programlama dili (örneğin C için) yukarıdaki tanımların anlamı nedir?

**Yanıt:** Aslında C sonsuz sayıda geçerli C programlarının oluşturduğu kümedir. Yani geçerli bir C programı yukarıdaki terminolojiye göre bir string’tir. C standartlarında dili oluşturan tüm üretimler listelenmiştir.

**Soru:** Yukarıdaki bilgilerin derleyici tasarımı ve gerçekleştirimi ile ilgisi nedir?

**Yanıt:** Derleyici tasarımını ve gerçekleştirimini anlayabilmek için daha teknik yaklaşımların kullanılması gerekmektedir. Bu nedenle yukarıdaki terminoloji ve biçimsel dillerle ilgili malzemeleri biz tasarım ve gerçekleştirim sürecinde kullanacağız. Ayrıca derleyici yazımında kullanılan araçların pek çoğu grameri bizden üretici gramer biçiminde istemektedir.

**3.1.6. BNF ve EBNF Notasyonları**

Üretici gramer fikri Noam Chomsky tarafından 1956’da ortaya atılmıştır. Fakat Chomsky bir bilgisayar bilimcisi değildi. Daha çok doğal diller ve biçimsel diller konusunda çalışmalar yapıyordu. Dünyanın ilk yüksek seviyeli programlama dili olan Fortan 1954-1957 yılları arasında geliştirildi. Fortan’ın tasarımı büyük ölçüde John Backus tarafından yapılmıştı. Fortran’dan sonra onu Algol dili (Algol 60) izledi. Algol de John Backus ve Peter Naur gibi kişilerin öncülüğünde geliştirildi. İşte Backus ve Naur birbirlerinden bağlantısız bir biçimde Chomsky’nin üretici gramerinden esinlenerek programlama dilleri için üretici gramer yöntemleri geliştirdiler. İkisinin ayrı ayrı geliştirdiği bu yöntemler birbirlerine de benziyordu. Bu yöntemler daha sonraları birleştirilerek BNF (Backus- Naur Form) notasyonu doğdu. Bugün programlama dillerinin resmi gramerleri BNF notasyonu ve onun türevleriyle ifade edilmektedir. BNF notasyonu ISO tarafından genişletilerek EBNF (Extended BNF) ismiyle standardize edilmiştir. (Kurs dokümanlarında bu standartları bulabilirsiniz). Ancak C, C++, Java, C# gibi dillerin sentaksları EBNF ile değil klasik BNF notasyonunun türevleriyle açıklanmış durumdadır.

**3.1.6.1. BNF Notasyonun Temel Özellikleri**

BNF notasyonu standardize edilmediği için birbirlerine benzeyen pek çok biçimi kullanılmaktadır. Notasyonun temel özellikleri şöyledir:

1) Üretimlerde açılacak ara sembollerden sonra ‘:’ karakteri yerleştirilir. Bunun yanına ya da aşağısına da açımdan elde edilecek hedef semboller yerleştirilmektedir. Örneğin:

*declaration:*

*declaration-specifiers init-declarator-listopt* **;**

2) Semboller son semboller (terminal symbols) ve ara semboller (non terminal symbols) olmak üzere ikiye ayrılmaktadır. Son semboller atomları (ya da alfabedeki elemanları) belirtir. Genellikle ara semboller italik olarak yazılırlar. Son semboller ise genellikle bold bir biçimde belirtilmektedir. Son sembollerin bold yerine tek tırnak içerisinde yazılması da yaygındır.

3) Bir ara sembolün seçenekleri ya ‘|’ sembolleriyle ya da alt alta belirtilir. Örneğin:

*Digit:*

‘0’

‘1’

‘2’

‘3’

‘4’

‘5’

‘6’

‘7’

‘8’

‘9’

Diğer bir yazım biçimi şöyle olabilirdi:

*Digit:* ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

4) Bir sembolün olup olmamasının isteğe bağlı olması genellikle alt indis olarak “opt” sözcüğü ile ifade edilmektedir. Örneğin:

*declaration:*

*declaration-specifiers init-declarator-listopt* **;**

Burada “declaration” “declarator-specifiers” ve isteğe bağlı bir “init-declararor-list”ten oluşmaktadır. Yani başka bir deyişle “declaration”da “init-declarator-list” bulunmak zorunda değildir.

Şimdi BNF için bazı örnekler verelim:

*decimal-constant:*

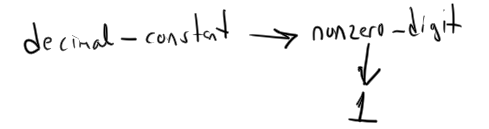
*nonzero-digit*

*decimal-constant digit*

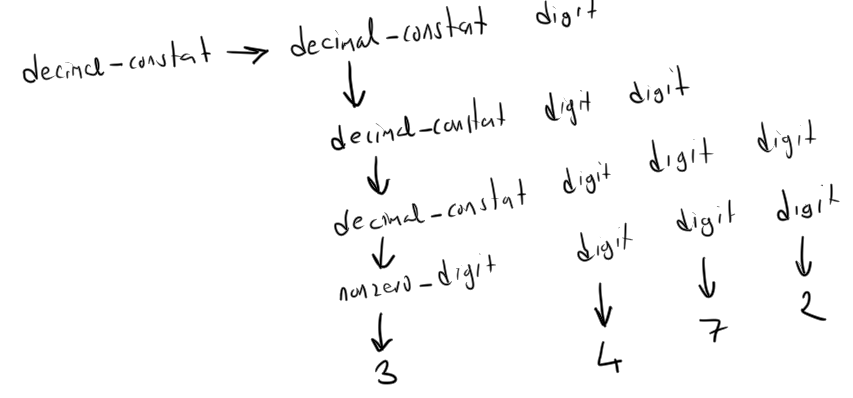
*nonzero-digit: '*1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

*digit: '*0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

Şimdi "decimal-constant" başlangıç ara sembolünden hareketle çeşitli açımlar yapalım:



Burada 1 sayısı açılımdan elde edilmiştir. Örneğin:



Burada 3472 sayısı elde edilmiştir. "decimal-constant" gramerinde başı sıfır ile başlamayan her türlü tamsayının elde edilebileceğine dikkat ediniz.

Şimdi aşağıdaki gramere bakalım:

*Exp:*

*Additive*

*Additive:*

‘id’ ‘+’ *Additive*

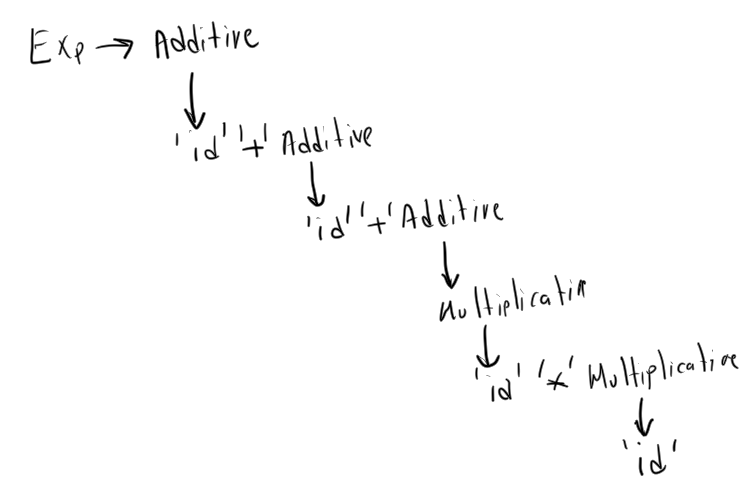
*Multiplicative*

*Multiplicative:*

‘id’ ‘\*’ *Multiplicative*

‘id’

Bu gramerde başlangıç sembolü “Expression” sembolüdür. Burada biraz tersten giderek şu çalışmayı yapalım. Yukarıdaki gramer 'id' + 'id' + 'id' \* 'id' açılımını yapabilir mi? Evet aşağıdaki gibi bir açılımla istenilen ifade elde edilebilir.



Pekiyi yukarıdaki "Expression" grameri ile 'id' \* 'id' + 'id' ifadesi açılabilir mi? Yanıt hayır! Bu grameri aşağıdaki gibi değiştirelim:

*Exp:*

*Additive*

*Additive:*

Multiplicative ‘+’ *Multiplicative*

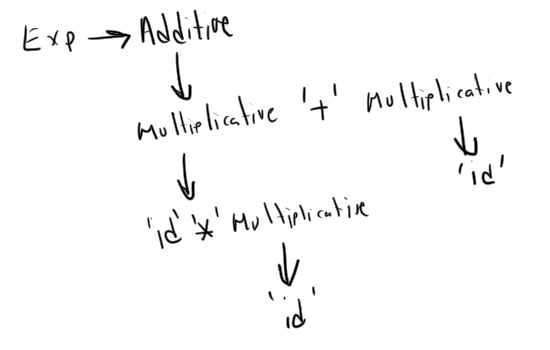
*Multiplicative*

*Multiplicative:*

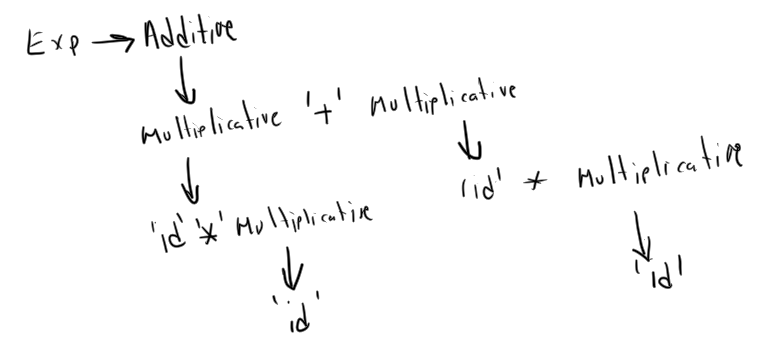
‘id’ ‘\*’ *Multiplicative*

‘id’

Şimdi bu gramerle ile 'id' \* 'id' + 'id' ifadesini açmaya çalışalım:



Şimdi de 'id' \* 'id' + 'id' \* 'id' ifadesini açmaya çalışalım:



Şimdi de 'id' + 'id' + 'id' ifadesini bu gramerle açmaya çalışalım. Açabilir miyiz? Yanıt hayır! Pekiyi grameri biraz daha değiştirelim:

*Exp:*

*Additive*

*Additive:*

Additive ‘+’ *Multiplicative*

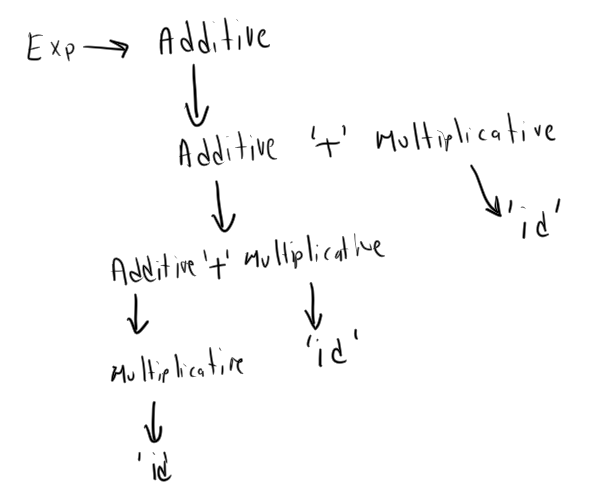
*Multiplicative*

*Multiplicative:*

‘id’ ‘\*’ *Multiplicative*

‘id’

Artık 'id' + 'id' + 'id' ifadesini açabiliriz:



Bu son gramerin aşağıdaki tüm ifadeleri açabileceğine dikkat ediniz:

'id'

'id' + 'id' \* 'id'

'id' \* 'id' \* 'id'

'id' + 'id' + 'id'

'id' \* 'id' + 'id' \* 'id'

Şimdi grameri biraz daha geliştirelirm:

*Exp:*

*Additive*

*Additive:*

Additive ‘+’ *Multiplicative*

*Additive* ‘-‘ *Multiplicative*

*Multiplicative*

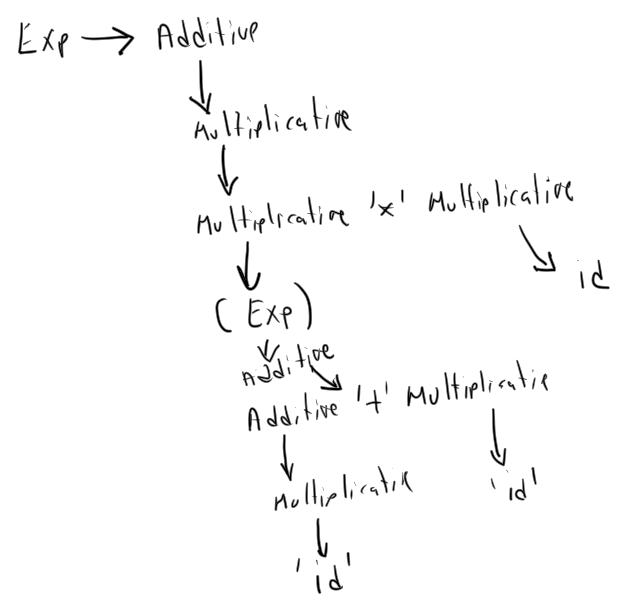
*Multiplicative:*

*Multiplicative* ‘\*’ *Multiplicative*

‘id’

‘(‘ Exp ’)’

Şimdi de ('id' + 'id') \* 'id' bir ifadeyi bu gramerle açmaya çalışalım:



Bu gramerle aşağıdaki ifadelerin hepsinin açılabileceğine dikkat ediniz:

'id'

'id' + 'id'

'id' \* ('id' + 'id')

('id' + 'id') \* ('id' + 'id')

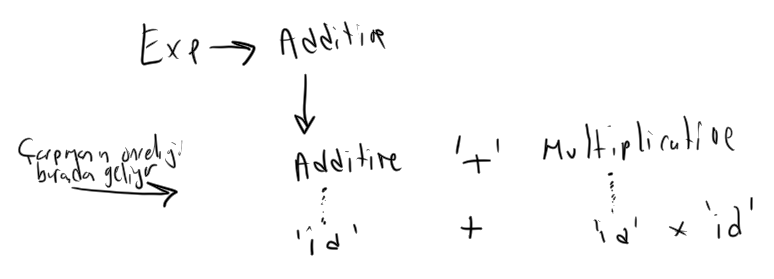
('id' \* ('id' + 'id')) \* ('id' + 'id')

Bu gramer programlama dillerindeki “ifade” kavramını bire bir yansıtmaktadır.

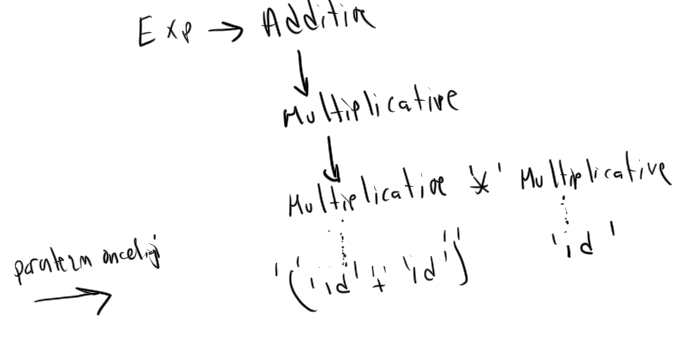
Programlama dillerindeki işlem önceliği ve operatörlerin öncelik tablosu aslında programcıların kolay anlaması için uyfurulan kavramlardır. Gerçekten de C ve C++ standartlarında operatör önceliği diye bir kavram yoktur. Gramerin kendisi zaten bizzat bu önceliği belirtmektedir. Şöyle ki:

'id' + 'id' \* 'id'

Bu ifade yukarıdaki gramere göre açıldığında 'id' ile 'id' \* 'id’nin toplanacağı anlaşılır. id + id ile id’nin çarpılması bu gramere göre zaten münkün değildir. Çünkü açım bu gramere göre ancak şöyle yapılabilir:



Parantezlerin önceliği de gramerin içerisinden çıkartılacak bir sonuçtur. Örneğin ('id' + 'id') \* 'id' ifadesinde açılım ancak şöyle olabilir:



Başka bir deyişle operatör önceliği zaten gramer tarafından dolaylı olarak belirtilmiş durumdadır. Ayrıca bir öncelik tablosuna gereksinin yoktur. Gerçekten de C ve C++ standartlarına baktığınızda operatör önceliği diye bir konunun olmadığını göreceksiniz. Çünkü örneğin 'id' + 'id' \* 'id' ifadesi 'id' + 'id' ile 'id’nin çarpımı biçiminde bu gramer tarafından açılamamaktadır. Yani bu ifade aslında yukarıdaki gramere göre şöyle açılmaktadır:

İ1: id \* id

İ2: id + İ1

C Programlam dilinin standartlarında belirtilen başlangıç sembolü “translation-unit” isimli semboldür:

*translation-unit:*

*external-declaration*

*translation-unit external-declaration*

Buradan "translation-unit" (yani kaynak dosya) sembolünün bir ya da birden fazla "external-declaration"dan oluştuğunu söyleyebiliriz. external-declaration sembolü de şöyle belirtilmiştir:

*external-declaration:*

*function-definition*

*declaration*

Görüldüğü gibi bir "external-decalaration" ya bir fonksiyon tanımlasından ya da bir bildirimden oluşmaktadır. Örneğin aşağıdaki bir C programı olsun:

int x;

void foo(void)

{

...

}

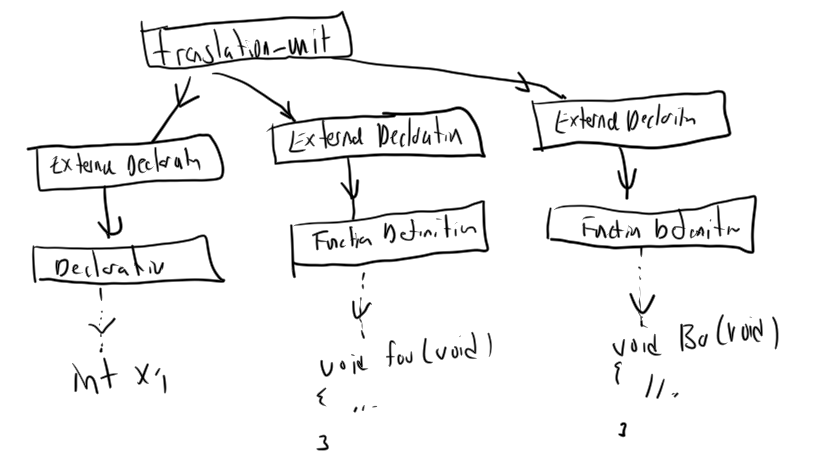
void bar(void)

{

...

}

Bu program gramerden şöyle elde edilebilir:



**3.1.6.2. EBNF Notasyonunun Temel Özellikleri**

Yukarıda da belirttiğimiz gibi BNF Notasyonu standardize edilmemişti. ISO BNF notasyonunu geliştirerek EBNF (Extended BNF) ismiyle standardize etti (ISO/IEC-14977: 1996). EBNF yazımı kolaylaştımrak için birtakım meta karakterler kullanmaktadır. Meta karakterler son sembol olmayan, özel anlama gelen karakterlerdir.

EBNF notasyonun temel özellikleri şunlardır:

1) Ara sembolü açıklamak için ‘=’ meta karakteri kullanılmaktadır.

2) Son semboller (terminal symbols) iki tırnak içerisinde belirtilirler. Ara sembol tanımlamasını bitirmek için ‘;’ meta karakteri yan yana gelen atomları ayırmak için ise ‘,’ meta karakteri kullanılmaktadır.

3) Seçenekler BNF'de olduğu gibi yine ‘|’ meta karakterleriyle belirtilirler. Örneğin:

digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;

4) ‘? ‘ metakarakteri tek bir ifadenin isteğe bağlı olduğunu, ‘\*’ meta karakteri ilgili ifadenin sıfır tane ya da daha fazla yineleneceğini, ‘+’ meta karakteri ise ilgili ifadenin bir ya da daha fazla yinelenceğini belirtmektedir.

5) Köşeli parantezler içerisindeki öğeler isteğe bağlı (optional) öğeleri belirtir. Gruplama için normal parantezler kullanılır.

5) Tekrarlamalar EBNF’de daha kolay bir biçimde küme parantezleriyle belirtilmektedir. Küme parantezleri içerisindeki öğeler bir ya da birden fazla kez yinelenebilir. Örneğin:

identifier = letter , { letter | digit | "\_" }

Burada "identifier", " letter" ve "digit" ara sembollerdir. Bir "identifer" tek bir "letter"dan oluşabilir. Ya da bir "letter" ile başlayıp bir "letter", "digit" ya da “\_” karakterlerinden istenildiği kadar yan yana getirilerek oluşturulabilir.

Aşağıdaki “en.wikipedia.org”den alınmış örneği inceleyiniz:

letter = "A" | "B" | "C" | "D" | "E" | "F" | "G"

| "H" | "I" | "J" | "K" | "L" | "M" | "N"

| "O" | "P" | "Q" | "R" | "S" | "T" | "U"

| "V" | "W" | "X" | "Y" | "Z" | "a" | "b"

| "c" | "d" | "e" | "f" | "g" | "h" | "i"

| "j" | "k" | "l" | "m" | "n" | "o" | "p"

| "q" | "r" | "s" | "t" | "u" | "v" | "w"

| "x" | "y" | "z" ;

digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;

symbol = "[" | "]" | "{" | "}" | "(" | ")" | "<" | ">"

| "'" | '"' | "=" | "|" | "." | "," | ";" ;

character = letter | digit | symbol | "\_" ;

identifier = letter , { letter | digit | "\_" } ;

terminal = "'" , character , { character } , "'"

| '"' , character , { character } , '"' ;

lhs = identifier ;

rhs = identifier

| terminal

| "[" , rhs , "]"

| "{" , rhs , "}"

| "(" , rhs , ")"

| rhs , "|" , rhs

| rhs , "," , rhs ;

rule = lhs , "=" , rhs , ";" ;

grammar = { rule } ;

Bu gramerden şu sonuçları çıkartabiliriz: Buradaki gramer “rule”lardan oluşmaktadır. Bir rule kabaca şu biçimdedir:

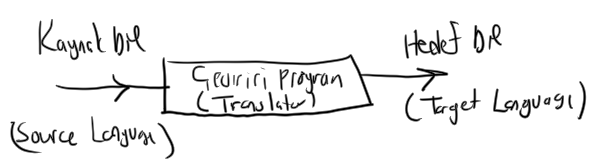
lhs = rhs;

lhs (left hand side) identifier olmak zorundadır. Ancak rhs çeşitli biçimlerde açılabilmektedir.

XML standartlarında (Ecma-357) gramer EBNF ile açıklanmıştır. Bu standartları gözden geçirerek EBNF notasyonu konusunda uzmanlaşabilirsiniz.

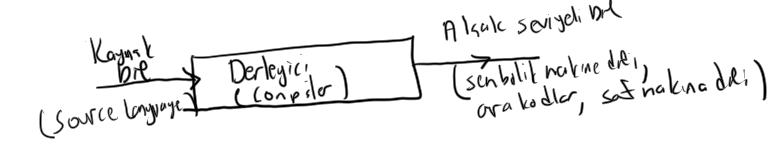
**3.2. Programlama Dilleri Üzerinde İşlemler Yapan Araçlar**

Bir programlama dilinde yazılmış olan bir programı eşdeğer olarak başka bir programlama diline çeviren programlara “çevirici programlar (translators)" denilmektedir.



Bir çevirici programda çevrilmek istenen programın diline “kaynak dil (source language)”, çevirme sonucunda elde edilen programın diline de “hedef dil (target language)” denilmektedir. Örneğin bir C# programını eşdeğer olarak VB.NET programına dönüştüren çevirici programın kaynak dili C#, hedef dili ise VB.NET’tir. Şüphesiz her dili her dile dönüştüren bir çevirici programın yazılabilmesi mümkün olmayabilir.

Bir çevirici programda hedef dil alçak seviyeli bir dilse (yani sembolik makine dili, ara kod ya da saf makine dili) böyle çevirici programlara “derleyici (compiler)” denilmektedir.



Gördüğünüz gibi bir çevirici programın derleyici biçiminde isimlendirilmesi kaynak dile değil hedef dile bağlıdır. Örneğin sembolik makine dilinde yazılmış programları da saf makine diline dönüştüren programlara derleyici denilmektedir. Benzer biçimde Java ve .NET platformlarında bu platformların arakodlarını gerçek makine kodlarına dönüştüren sistemler de derleyici olarak tanımlanırlar. Bu tür derleyicilere özel olarak “Just In Time Compiler” denilmektedir. Özel olarak sembolik makine dilinde yazılmış programı saf makine diline dönüştüren programlara “assembler” da denilmektedir. “Assembler” terimi aslında sembolik makine dili derleyicisi anlamına geliyorsa da bu terim yanlış kullanımlarla "sembolik makine dilinin (assembly language)” kendisini de anlatır hale gelmiştir.

Alçak seviyeli dilleri girdi olarak alıp bunları yüksek seviyeli dillere dönüştüren çevirici programlara “decompiler” denilmektedir. "Decompiler"ların işlevsel olarak derleyicilerin tam tersi bir işlemi yaptıklarına dikkat ediniz. Örneğin .NET’in arakodunu yeniden C#’a dönüştüren pek çok “decompiler” vardır (Reflector, ILSpy, Salamander, Dis# gibi). Maalesef saf makine dillerinen yüksek seviyeli dillere etkin dönüştürme yapan "decompiler"lar etkin biçimde yazılamamaktadır.

Saf makine dilinden sembolik makine dillerine dönüştürme yapan çevirici programlara “disassembler” denilmektedir. Örneğin çalıştırılabilen bir programı sembolik makine dilinde bize gösteren pek çok "disassember" vardır. Ancak “disassembler” terimi daha çok yalnızca makine komutlarının dönüşümünü yapan programlar için tercih edilmektedir. Aslında pek çok kaynak saf makine dilinden belli bir sembolik makine dili çıktısı üreten programları belirtmek için yine “decompiler” terimini tercih etmektedir.

Bazen derleme işleminin yapıldığı makinenin işlemci ailesi ile derleme sonucunda üretilen kodun çalıştırılacağı işlemci ailesi birbirinden farklı olabilmektedir. İşte bu biçimde çalıştırıldığı ortamdaki işlemcinin kodunu değil de başka bir işlemcinin kodunu üreten derleyicilere “çapraz derleyiciler (cross compilers)” denilmektedir. Örneğin kişisel bilgisayarlarımızda Intel ailesi işlemciler kullanılıyor. Biz böyle bir bilgisayarda cep telefonlarında çalıştırılmak üzere ARM kodu üreten bir derleyici kullanıyorsak bu derleyici bir çapraz derleyicidir. Örneğin mikrodenetleyici kodları hemen her zaman çapraz derleyicilerle derlenmektedir.

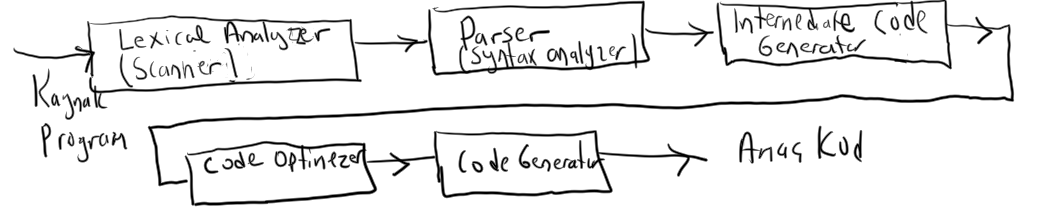
Yorumlayıcılar (interpreters) çevirici programlar değillerdir. Yani yorumlayıcılar kod üretmezler. Bir yorumlayıcı kaynak kodu okur, onu hiç kod üretmeden doğrudan çalıştırır.

**Anhtar Notlar:** “Translator” İngilizce çevirmen anlamına gelmektedir. Yazılı metni çeviren kişilere çevirmen denir. Interpreter ise İngilizce “mütercim tercüman” anlamına gelmektedir. Konuşmayı çevirenlere mütercim tercüman denir. İngilizce “interpreter” aslında mütercim tercüman anlamından yazılım dünyasına uyarlanmıştır. Fakat biz “interpret” sözcüğünün Türkçe karşılığı için "mütercim tercüman" yerine “yorumlayıcı” sözcüğünü kullanacağız.

**3.3. Derleme ve Yorumlama İşleminin Aşamaları**

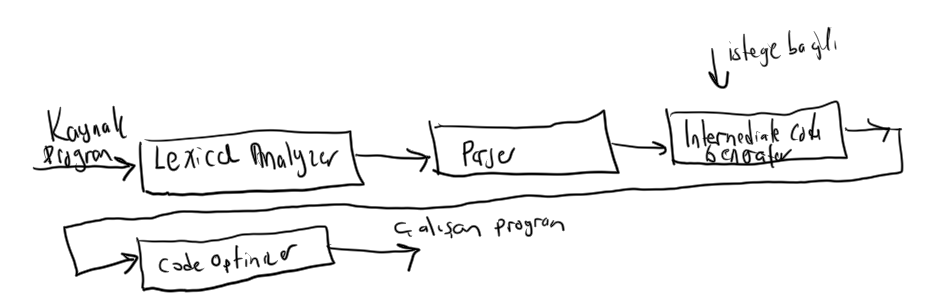
Hem derleyiciler hem de yorumlayıcıların gerçekleştirimlerinde birtakım ortak aşamalar vardır. Ancak yorumlayıcılar kod üretimi yapmadıkları için onların gerçekleştirilmeleri derleyicilere göre daha kolaydır.

Bir derleyici tipik olarak şu aşamalardan geçilerek gerçekleştirilmektedir:



Kaynak kod derleyicinin “lexical analiz (lexical anaylzer)”, “tarayıcı (scanner)” ya da “atom ayrıştırıcısı (tokenizer)” denilen modülü tarafından ele alınır. Bu modülün görevi kaynak kodu atomlarına ayırmaktır. Bu modülden sonra “parser” modülü devreye girer. "Parser" modülüne “syntax analyzer” da denilmektedir. Parser modülü atomları girdi olarak alır ve bu atomların dizilişlerinin dilin gramerine uygun olup olmadığını denetler. Parser modülü ürün olarak “parse ağacı (parse tree)” denilen bir ağaç oluşturmaktadır. “Parse ağacı” kaynak kodun işlenebilir bir veri yapısı haline dönüştürülmüş biçimidir. Yani program artık bir yazı olmaktan çıkmış bir veri yapısı olarak ifade edilmiştir. Sonraki modül parse ağacını özyinelemeli biçimde dolaşır ve ağaçtaki elemanlar için arakodlar (intermediate codes) üretir. Arakod gerçek makine kodu değildir. Onu temsil eden bir geçiş kodudur. Bu işlemi yapan modüle “ara kod üreticisi (intermediate code generator)” denilmektedir. Daha sonra bu ara kodlar optimize edilir. Bu modül de “kod eniyileyicisi (code optimizer)” olarak isimlendirilmektedir. Nihayet optimize edilmiş arakodlardan gerçek makine kodları oluşturulur. Bu modül de “kod üreticisi (code generator)” olarak isimlendirilmektedir.

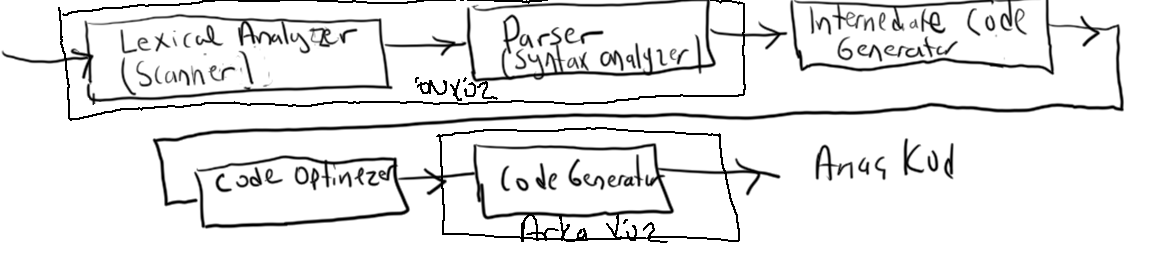
Yorumlayıcılar da benzer aşamalardan geçerek gerçekleştirilirler. Aslında bir yorumlayıcı “kod üretici (code generator)” kısmı olmayan bir derleyici gibi düşünülebilir. Bazı yorumlayıcılar hiç arakod üretmezler, doğrudan parse ağacı üzerindeişlem yaparlar.



Derleyicilerin kaynak program üzerinde işlem yaptığı modüllere “ön yüz (frontend)”, makine kodunu ürettiği modüllere “arka yüz (back end)” ve aradaki diğer modüllere de “orta yüz (middle end)” denilmektedir. Ön yüz kaynak dile, arka yüz ise hedef dile bağlıdır. Derleyicileri yazanlar önyüzlerin çıktılarını (parse ağacını vs.) standart hale getirmeye çalışırlar. Böylece derleyicinin port edilmesi kolaylaşır. Şöyle ki: Bir firma düşünelim. Bu firma N tane dili M tane makine dili için derleyecek derleyiciler yazmak istesin. Normalde bunun için kaç derleyicinin yazılması gerekir? Yanıt: N \* M tane değil mi? Halbuki firma N tane dil için önyüz, M tane dil için de arka yüz yazarsa toplamda N + M tane faaliyetle bunları birleştirebilir.

Örneğin biz bir dil tasarlamış olalım. Ancak bunun derleyicisi için gcc derleyicisinden faydalanmak isteyelim. Mademki gcc işin önemli kısmını zaten yapmaktadır. O halde biz gcc için kendi dilimizi atomlarına ayıran ve onu parser modülüne veren bir önyüz yazabiliriz. Bu durumda kod optimizasyonunu ve kod üretimini gcc’nin zaten var olan modülleri yapacaktır. İşte bu faaliyete gcc için “frontend” yazma faaliyeti denilmektedir. Şimdi biz yeni bir işlemci için C derleyicisi yazmak isteyelim. Ve bunun için yine gcc derleyicisinden faydalanmak isteyelim. Bu durumda gcc’nin C frontend’i doğrudan kullanılabilir. Fakat bizim gcc için bir hedeflediğimiz işlemcinin kodunu üreten bir "arkayüz (backend)” yazmamız gerekir.

Derleyicilerin ya da yorumlayıcıların “Lexical analiz” ve “Parser” modülleri onların önyüzlerini oluşturmaktadır. “kod üretici modülleri ise onların arka yüzünü oluşturur. Geri kalan kısımlar modüller ise orta yüze ilişkindir. Yorumlayıcılar kod üretmimi yapmakdıkları için bunların arkayüzleri yüzlerinin olmadığını yeniden anımsatalım:



Bu konuda sıkça sorulan sorular ve yanıtları şöyledir:

**Soru:** Derleyicilerde ve yorumlayıcılarda önyüz (frontend) nedir?

**Yanıt:** Önyüz derleyicilerin ve yorumlayıcıların kaynak dil üzerinde işlem yapan modüllerdir. Kaynak kodun atomlarına ayrılması ve parse edilmesi doğrudan kaynak dilin sentaksı ve semantiği ile ilgilidir. Bu nedenle “lexical analiz” ve “parser” modülleri önyüze ilişkin modüllerdir.

**Soru:** Derleyicilerde arkayüz nedir?

**Yanıt:** Arkayüz hedef koda yünelik işlem yapan modülden oluşur. Bu da tipik olarak “kod üretici (code generator)” modülüdür. Önyüzü yazmak için kaynak dili, arkayüzü yazmak için ise hedef dili bilmek gerekir.

**Soru:** Derleyicilerin gerçekleştirme aşaması bakımından yorumlayıcılardan ne farkı vardır?

**Yanıt:** Derleyiciler arkayüze sahiptir halbuki yorumlayıcılar kod üretmedikleri için sahip değildir.

**Soru:** Derleme ve yorumlama sürecindeki arakod kavramı nedir?

**Yanıt:** Arakod gerçek hedef kodu temsil eden fakat hedef koddan bağımsız bir koddur. Optimizasyonların çoğu arakodlar üzerinde yapılmaktadır. Böylece derleyiciler port edilirken arakod değişmeyeceği için optimizasyon işlemleri büyük ölçüde hedef koddan bağımsız hale getirilmiş olur.

**Soru:** Bir derleyiciyi “port etmek” ne anlama gelmektedir?

**Yanıt:** Belli bir mikroişlemci için hedef kod üreten derleyiciyi başka bir işlemci için hedef kod üretecek hale getirme sürecine “port etmek” denilmektedir. Örneğin gcc derleyicileri yalnızca Intel ailesi için değil pek çok mikroişlemci ailesi için kod üretir durumdadır. Ancak port etmek önyüz değiştirme faaliyeti için de kullanılabilmektedir.

**3.4. Kaynak Kodun Atomlarına Ayrılması (Lexical Analysis / Scanning / Tokenizing)**

Derleyicilerin ve yorumlayıcıların ilk aşaması kaynak kodun atomlarına ayrılmasıdır. Bu sürece İngilizce “lexical analysis”, “scanning” ya da “tokenizing” denilmektedir. Derleyicilerde bu süreci gerçekleştiren modüller de benzer biçimde “lexical analyzer”, “scanner” ya da “tokenizer” biçiminde isimlendirilmektedir. Biz kursumuzda bu sürece yarı İngilizce yarı Türkçe "lexical analiz" diyeceğiz. Derleyiciler ve yorumlayıcılar tarafından kaynak kod yalnızca “lexical analiz” aşamasında okunmaktadır. Dolayısıyla bu aşama bir dosya işlemi gerektirdiğiiçin bu işlemlerin yürütülme biçimi de derleyici ya da yorumlayıcının performansını etkilemektedir.

“Lexical analiz” modülünün aslında tek bir fonksiyondan oluştuğunu söyleyebiliriz. Bu fonksiyon kaynak kodda kalınan yerden sonraki ilk atomun karakterlerini ve türünü bize verir. Bu işlemi yapan fonksiyonun arayüzü şöyle olabilir:

char g\_token[MAX\_TOKEN];

int GetNextToken(void);

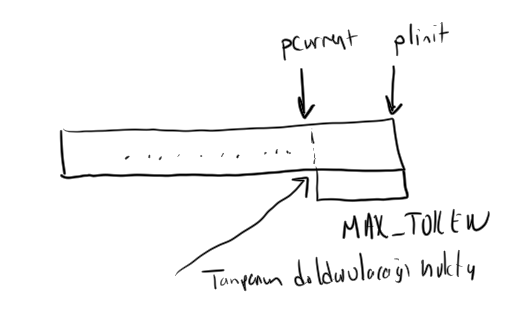
Burada GetNextToken bize kalınan yerden sonraki ilk atomu veren fonksiyondur. Bu fonksiyon atomun yazısını g\_token isimli char türden diziye yerleştirip atomun türüyle geri dönmektedir. Tipik olarak atom türlerinin her biri bir sayıyla temsil edilmektedir. Tabii yukarıdaki verdiğimiz arayündeki fonksiyonların ve global değişkenin isimlendirme ve harflendirme (capitalization) biçimine takılmayınız. Bu isimlendirme biçimlerini istediğiniz gibi değiştirebilirsiniz. Bu arayüzde global bir nesnenin kullanılması da sizi şaşırtmasın. Bildiğiniz gibi global nesnelere erişim bilgisayar zamanı bakımından daha hızlıdır. Bu nedenle derleyici aşamalarının etkin bir biçimde gerçekleştirilmesi için global nesnelerden faydalanılmaktadır. Tabii aşağıdaki gibi bir arayüz de söz konusu olabilirdi:

int GetNextToken(char \*token);

Bu arayüzde GetNextToken fonksiyonu sıradaki atomu parametresiyle aldığı adrese yerleştirmektedir. Fakat biz daha çok birinci arayüzü tercih edeceğiz.

**3.4.1. Atomlara Ayırma İşlemi İçin Tamponlama Mekanizması**

Lexical analiz modülü kaynak kodu mümkün olduğunca etkin bir biçimde okumalıdır. Oysa karakterleri fgetc ya da getc gibi bir fonksiyonla tane tane okumak (her ne kadar standart C fonksiyonları tamponlama yapıyorsa da) göreli olarak zaman kaybına yol açar. Bu nedenle bir tamponlama mekanizmasının kullanılması tercih edilir. Yani dosyanın karakterleri bir tampona okunur, lexical analiz modülü de bu tampondan karakterleri alır. Tampondaki karakterler bitince tampon yeniden doldurur. Ancak lexical analiz modülü atomlarına ayırma işlemi sırasında atomu tespit edebilmek için sonraki karakterlere de bakmak zorunda kalabilmektedir. İşte bir atomun bazı karakterlerinin tamponda olması bazılarının olmaması gerçekleştirimi zorlaştırmaktadır. Bu nedenle tamponda en az bir atomun uzunluğu kadar karakterin bulundurulması yoluna gidilir. Yani başka bir deyişle tamponun doldurulması tamponun sonuna gelince değil sondan MAX\_TOKEN gibi bir seviyeye gelince yapılır.



Şekildeki pcurrent tamponda kalınan yeri, plimit de tamponun sonunu göstermektedir. Tamponun doldurulması şöyle bir kontrolle yapılabilir:

if (plimit - pcurrent < MAX\_TOKEN) {

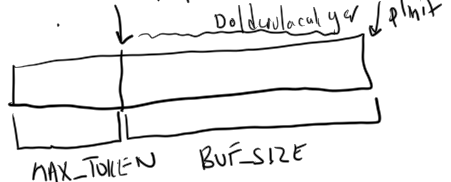
FillBuffer();

...

}

Pekiyi buaradaki MAX\_TOKEN hangi değerde olmalıdır? Örneğin C’de bir atomun maksimum uzunluğu nedir? İşte pek çok atomun uzunluğu bir ya da birkaç karakterdir. Ancak istisna olarak değişkenlerin (identifer), sabitlerin (literals), iki tırnak ifadelerinin (strings) ve yorum (remark) kısımlarının uzunlukları çok fazla olabilmektedir. Bu dört istisna durum özel durumlar olarak değerlendirilebilir. Dolayısıyla MAX\_TOKEN bu üç atom grubununun dışındaki atomların maksimum uzunluğudur.

Pekiyi tamponun doldurulması nasıl yapılmalıdır? Burada kullanılan tampon bir kuyruk sistemine benzetilebilir. Dolayısıyla tamponun gerçekleştirimi döngüsel bir kuyruk sistemi ile yapılabilir. Ancak döngüsel kuyruk sistemlerinde her karakterde tamponun sonuna gelindi mi diye bir kontrolün yapılması gerekmektedir. İşte bu kontrolden kurtulmak için genellikle tamponun sonındaki MAX\_TOKEN kadar karakter tamponun başına kopyalanıp oradan devam edilir. Pekiyi okuma ne kadar uzunlukta yapılacaktır? Dosya okumalarının belli değerlerin katları (örneğin sektörlerin) kadar yapılması okuma verimliliğini yükseltebilmektedir. Dolayısıyla okuma miktarı BUF\_SIZE ile temsil edilirse bizim bu kadarlık okumayı yapabilmemiz için gerçek tamponun bundan MAX\_TOKEN kadar daha büyük olması gerekir.



Burada bir noktaya dikkatinizi çekmek istiyoruz.-Dosyadaki karakterler BUF\_SIZE’ın katlarından küçük olabilir. Bu durumda son okuma tamponu tamamen doldurmayacaktır.

Karakterler tampondan tek tek okurken tamponun ya da dosyanın sonuna gelinip gelinmediği kontrol edilmelidir. İşte benzer kontrollerin ayrı ayrı yapılması yerine onlar birleştirebilir. Şöyle ki: Tamponun ve dosyanın sonu için tampona ‘\n’ karakteri yerleştirilir. Bu ‘\n’ karakteri okunduğunda ya yeni bir satıra geçilmiştir ya da dosyanın sonuna gelinmiştir. Böylece sona gelindi mi kontrolü yalnızca ‘\n’ karakteri görüldüğünde yapılır. Zaten pek çok lexical analiz modülü atomlara ayırma işlemi sırasında hata mesajları için satır numaralarını saklar. Bu durumda ‘\n’ karakteri görüldüğünde ya tamponun sonuna, ya dosyanın sonuna ya da satırın sonuna gelinmiştir.

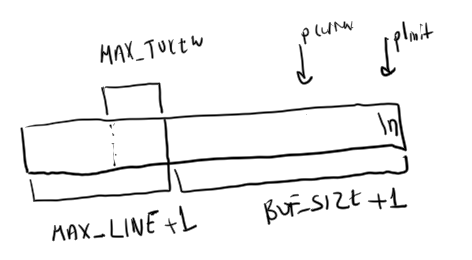
Yukarıda programlama dillerindeki pek çok atomun birkaç karakterden oluştuğunu söyledik ve bu atomların maksimum uzunluğunu MAX\_TOKEN değeri ile temsil ettik. Ancak yine yukarıda MAX\_TOKEN değerinden daha uzun olabilecek üç tür atomun olduğunu da belirttik. Bunlar değişkenler (identifiers), sabitler, iki tırnak ifadeleri (strings) ve yorumlama (remark) alanlarıydı. Şimdi bu atomların elde edilmesi için nasıl bir tampon strateji izleneceği üzerinde duralım. Değişkenler ve iki tırnak ifadeleri pek çok dilde en fazla bir satırın uzunluğu kadar olabilmektedir. (Çünkü satırın sonunda ‘\n’ karakteri vardır ve bu karakter zaten atomu sonlandırmaktadır.) İşte derleyicilerin çoğu işlemleri kolaylaştırmak için kaynak koddaki maksimum satır uzunluğu konusunda bir ön belirleme yapmaktadır. Biz de burada maksimum satır uzunluğunun MAX\_LINE kadar olduğunu varsayalım. Böylece eğer boşluklar atıldıktan sonraki ilk karakter alfabetik bir karakterse, sayısal bir karakterse ya da iki tırnak karakteriyse gelmekte olan atom en kötü olasılıkla MAX\_LINE kadar olabilir. Bu üç durumda tamponun doldurulma noktası MAX\_TOKEN değil MAX\_LINE olmalıdır. Satırın sonunda ve dosyanın sonunda ‘\n’ karakterleri bulunacağından tamponun da maksimum uzunluğu MAX\_LINE + 1 + BUFSIZE + 1 kadar olacaktır:

if (plimit - pcurrent < MAX\_LINE) {

FillBuffer();

...

}



Yorum alanlarının koddan atılması sırasında tamponun yine satır uzunluğu kadar doldurulması uygun olabilir. Tabii yorum alanları birden fazla satırdan oluşlabilmektedir. Buy durum bir istisna olarak ele alınabilir.

**3.4.2. Lexical Analiz İşleminin Algoritmik Yapısı**

Lexical analiz işleminde sıradaki atomun türü ve yazısal temsili tipik olarak şöyle bir algoritmik yapıyla gerçekleştirilmektedir:

1) Boşluk karakterleri pek çok dilde atom ayıracı olarak kullanıldığından dolayı önce boşluk klarakterleri geçilir. Boşluk karakterlerinin geçilmesi “boşluk karakteri olduğu sürece ilerleme yapan” bir while döngüsü ile gerçekleştirilebilir.

2) İlk boşluk karakteri olmayan karakter bulunır ve switch içerisine sokulur. Bu ilk karakter atomun türü hakkında bize bit ip ucu vermektedir. Bundan sonra duruma göre bu karakterin yanındaki karakterlere bakılarak atomun türü ve onu oluşturan karakterler net olarak belirlenir.

Örneğin C için bir lexical analiz modülü yazacak olalım. g\_cp göstericisi tampondaki son kalınan yeri belirtiyor olsun. Biz oradaki karakterin ‘+’ karakteri olduğunu düşünelim. Bu noktada biz henüz bunun + operatörü olduğu sonucunu çıkaramayız. Pekala bu operatör ++ ya da += operatörü de olabilir. Bunu belirlemek için bizim ‘+’ karakterini gördüğümüzde onun yanındaki karakterlere de bakmamız gerekir.

switch (\*g\_cp++) {

...

case ‘+’:

if (\*g\_cp == ‘+’)

return OPERATOR\_PLUS\_LUS;

if (\*g\_cp == ‘=’)

return OPERATOR\_PLUS\_EQUAL;

return OPERATOR\_PLUS;

...

}

İşte lexical analiz işlemi “boşlukları atıp sıradaki karaktere bak, duruma göre onun yanındakilere de bakarak atomu oluşturan karakterleri ve atomun türünü tespit et” biçiminde bir algoritmik yapıya sahiptir. (Bu tür parse işlemine LL(k) tipi parse işlemi denilmektedir. Bu konu “parse” işlemlerinde ele alınacaktır.)

Kaynak kodu tararken karakterlerin türlerini belirlemek için bazı karşılaştırma işlemleri gerekir. Örneğin bir karakterterin boşluk karakteri olup olmadığı aşağıdaki gibi bir if deyimiyle belirlenebilir:

if (ch == ' ' || ch == '\t' || ch == '\n' || ch = '\v') {

...

}

Ancak lexical analiz modüllerinde işlemleri hızlandırmak için genellikle bir “lookup” tablosundan faydalanılabilmektedir. Şöyle ki: 256 elemanlı bir karakter dizisi oluşturulur. Bu karakter dizisinin karakter kodlarına karşı gelen elemanlarına bitsel olarak özellikler atanır. Sonra da tek bir bitsel işlemle karakterin türü belirlenir. Örneğin böyle bir “lookup” tablosu aşağıdaki gibi oluşturulabilir:

enum CHAR\_CLASS {

BLANK = 0x01,

NEWLINE = 0x02,

ALPHA = 0x04,

DIGIT = 0x08,

OCTAL = 0x10,

HEX = 0x20,

OTHER = 0x40

};

static unsigned char g\_cmap[256] = {

/\* 000 nul \*/ 0,

/\* 001 soh \*/ 0,

/\* 002 stx \*/ 0,

/\* 003 etx \*/ 0,

/\* 004 eot \*/ 0,

/\* 005 enq \*/ 0,

/\* 006 ack \*/ 0,

/\* 007 bel \*/ 0,

/\* 010 bs \*/ 0,

/\* 011 ht \*/ BLANK,

/\* 012 nl \*/ NEWLINE,

/\* 013 vt \*/ BLANK,

/\* 014 ff \*/ BLANK,

/\* 015 cr \*/ 0,

/\* 016 so \*/ 0,

/\* 017 si \*/ 0,

/\* 020 dle \*/ 0,

/\* 021 dc1 \*/ 0,

/\* 022 dc2 \*/ 0,

/\* 023 dc3 \*/ 0,

/\* 024 dc4 \*/ 0,

/\* 025 nak \*/ 0,

/\* 026 syn \*/ 0,

/\* 027 etb \*/ 0,

/\* 030 can \*/ 0,

/\* 031 em \*/ 0,

/\* 032 sub \*/ 0,

/\* 033 esc \*/ 0,

/\* 034 fs \*/ 0,

/\* 035 gs \*/ 0,

/\* 036 rs \*/ 0,

/\* 037 us \*/ 0,

/\* 040 sp \*/ BLANK,

/\* 041 ! \*/ OTHER,

/\* 042 " \*/ OTHER,

/\* 043 # \*/ OTHER,

/\* 044 $ \*/ 0,

/\* 045 % \*/ OTHER,

/\* 046 & \*/ OTHER,

/\* 047 ' \*/ OTHER,

/\* 050 ( \*/ OTHER,

/\* 051 ) \*/ OTHER,

/\* 052 \* \*/ OTHER,

/\* 053 + \*/ OTHER,

/\* 054 , \*/ OTHER,

/\* 055 - \*/ OTHER,

/\* 056 . \*/ OTHER,

/\* 057 / \*/ OTHER,

/\* 060 0 \*/ DIGIT|OCTAL,

/\* 061 1 \*/ DIGIT|OCTAL,

/\* 062 2 \*/ DIGIT|OCTAL,

/\* 063 3 \*/ DIGIT|OCTAL,

/\* 064 4 \*/ DIGIT|OCTAL,

/\* 065 5 \*/ DIGIT|OCTAL,

/\* 066 6 \*/ DIGIT|OCTAL,

/\* 067 7 \*/ DIGIT|OCTAL,

/\* 070 8 \*/ DIGIT,

/\* 071 9 \*/ DIGIT,

/\* 072 : \*/ OTHER,

/\* 073 ; \*/ OTHER,

/\* 074 < \*/ OTHER,

/\* 075 = \*/ OTHER,

/\* 076 > \*/ OTHER,

/\* 077 ? \*/ OTHER,

/\* 100 @ \*/ 0,

/\* 101 A \*/ ALPHA|HEX,

/\* 102 B \*/ ALPHA|HEX,

/\* 103 C \*/ ALPHA|HEX,

/\* 104 D \*/ ALPHA|HEX,

/\* 105 E \*/ ALPHA|HEX,

/\* 106 F \*/ ALPHA|HEX,

/\* 107 G \*/ ALPHA,

/\* 110 H \*/ ALPHA,

/\* 111 I \*/ ALPHA,

/\* 112 J \*/ ALPHA,

/\* 113 K \*/ ALPHA,

/\* 114 L \*/ ALPHA,

/\* 115 M \*/ ALPHA,

/\* 116 N \*/ ALPHA,

/\* 117 O \*/ ALPHA,

/\* 120 P \*/ ALPHA,

/\* 121 Q \*/ ALPHA,

/\* 122 R \*/ ALPHA,

/\* 123 S \*/ ALPHA,

/\* 124 T \*/ ALPHA,

/\* 125 U \*/ ALPHA,

/\* 126 V \*/ ALPHA,

/\* 127 W \*/ ALPHA,

/\* 130 X \*/ ALPHA,

/\* 131 Y \*/ ALPHA,

/\* 132 Z \*/ ALPHA,

/\* 133 [ \*/ OTHER,

/\* 134 \ \*/ OTHER,

/\* 135 ] \*/ OTHER,

/\* 136 ^ \*/ OTHER,

/\* 137 \_ \*/ ALPHA,

/\* 140 ` \*/ 0,

/\* 141 a \*/ ALPHA|HEX,

/\* 142 b \*/ ALPHA|HEX,

/\* 143 c \*/ ALPHA|HEX,

/\* 144 d \*/ ALPHA|HEX,

/\* 145 e \*/ ALPHA|HEX,

/\* 146 f \*/ ALPHA|HEX,

/\* 147 g \*/ ALPHA,

/\* 150 h \*/ ALPHA,

/\* 151 i \*/ ALPHA,

/\* 152 j \*/ ALPHA,

/\* 153 k \*/ ALPHA,

/\* 154 l \*/ ALPHA,

/\* 155 m \*/ ALPHA,

/\* 156 n \*/ ALPHA,

/\* 157 o \*/ ALPHA,

/\* 160 p \*/ ALPHA,

/\* 161 q \*/ ALPHA,

/\* 162 r \*/ ALPHA,

/\* 163 s \*/ ALPHA,

/\* 164 t \*/ ALPHA,

/\* 165 u \*/ ALPHA,

/\* 166 v \*/ ALPHA,

/\* 167 w \*/ ALPHA,

/\* 170 x \*/ ALPHA,

/\* 171 y \*/ ALPHA,

/\* 172 z \*/ ALPHA,

/\* 173 { \*/ OTHER,

/\* 174 | \*/ OTHER,

/\* 175 } \*/ OTHER,

/\* 176 ~ \*/ OTHER,

};

Böylece kaynak koddan ch karakteri çekildiğinde bu karakterin ASCII tablosundaki sıra numarası bu diziye indeks yapılacak ve g\_cmap[ch] değeri de ilgili türlerle “bit and” işlemine sokulacaktır. Örneğin boşlukları g\_cp göstericisi tamponda kaynak koddaki kalınan yeri gösteriyor olsun. Boşluk karakterlerini geçmek için aşağıdaki gibi bir döngü oluşturulabilir:

while (g\_cmap[\*g\_cp] & BLANK)

++g\_cp;

Görüldüğü gibi karakter haritasının oluşturulmasının amacı atomun türünü hızlı bir biçimde belirlemektir.

Şimdi aşağıdaki gibi basit bir dili atomlarına ayrırmak isteyelim:

*Small-Lang*:

Expression

Expression Small-Lang

*Expression*:

*identifier* ‘=’ Additive ‘;’

*Additive*:

*Additive* ‘+’ *Multiplicative*

*Additive* ‘-‘ *Multiplicative*

Multiplicative

*Multiplicative*:

*Multiplicative* ‘\*’ *Multiplicative*

*Multiplicative* ‘/’ *Multiplicative*

*identifier*

‘(‘ *Expression* ’)’

*Identifier*:

*Alfa*

*AlfaNumeric* *Identifer*

*Alfa*:

‘a’ | ‘b’ .....

*AlfaNumeric*:

Alfa | 0 | 1 ....

Bu dilin birkaç elemanı şöyle olabilir:

a = b \* c \* d;

a = (b + c) \* d;

a = b + c – d / e;

Bütün programın global bir g\_prog isimli bir dizide bulunduğunu ve tamponlama yapılmadığını düşünelim. g\_cp göstericisi de işin başında bu dizinin başlangıcını gösteriyor olsun. Gramerde gösterilmemiş olsa da atomlar arasında istenildiği kadar boşluk karakterlerinin bulunabildiğini varsayalım. Bu koşullar altında atomlarına ayırma işlemi aşağıdaki gibi bir kodla yapılabilir:

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>

/\* Token Types \*/

#define TOKEN\_IDENTIFIER 1

#define TOKEN\_OPERATOR\_PLUS 2

#define TOKEN\_OPERATOR\_MINUS 3

#define TOKEN\_OPERATOR\_MULTIPLY 4

#define TOKEN\_OPERATOR\_DIVIDE 5

#define TOKEN\_LEFT\_PARANTHESIS 6

#define TOKEN\_RIGHT\_PARANTHESIS 7

#define TOKEN\_ASSIGNMENT 8

#define TOKEN\_SEMICOLON 9

#define MAX\_TOKEN 512

#define BLANK 0x01

#define NEWLINE 0x02

#define ALPHA 0x04

static unsigned char g\_cmap[256] = {

/\* 000 nul \*/ 0,

/\* 001 soh \*/ 0,

/\* 002 stx \*/ 0,

/\* 003 etx \*/ 0,

/\* 004 eot \*/ 0,

/\* 005 enq \*/ 0,

/\* 006 ack \*/ 0,

/\* 007 bel \*/ 0,

/\* 010 bs \*/ 0,

/\* 011 ht \*/ BLANK,

/\* 012 nl \*/ BLANK | NEWLINE,

/\* 013 vt \*/ BLANK,

/\* 014 ff \*/ BLANK,

/\* 015 cr \*/ 0,

/\* 016 so \*/ 0,

/\* 017 si \*/ 0,

/\* 020 dle \*/ 0,

/\* 021 dc1 \*/ 0,

/\* 022 dc2 \*/ 0,

/\* 023 dc3 \*/ 0,

/\* 024 dc4 \*/ 0,

/\* 025 nak \*/ 0,

/\* 026 syn \*/ 0,

/\* 027 etb \*/ 0,

/\* 030 can \*/ 0,

/\* 031 em \*/ 0,

/\* 032 sub \*/ 0,

/\* 033 esc \*/ 0,

/\* 034 fs \*/ 0,

/\* 035 gs \*/ 0,

/\* 036 rs \*/ 0,

/\* 037 us \*/ 0,

/\* 040 sp \*/ BLANK,

/\* 041 ! \*/ 0,

/\* 042 " \*/ 0,

/\* 043 # \*/ 0,

/\* 044 $ \*/ 0,

/\* 045 % \*/ 0,

/\* 046 & \*/ 0,

/\* 047 ' \*/ 0,

/\* 050 ( \*/ 0,

/\* 051 ) \*/ 0,

/\* 052 \* \*/ 0,

/\* 053 + \*/ 0,

/\* 054 , \*/ 0,

/\* 055 - \*/ 0,

/\* 056 . \*/ 0,

/\* 057 / \*/ 0,

/\* 060 0 \*/ 0,

/\* 061 1 \*/ 0,

/\* 062 2 \*/ 0,

/\* 063 3 \*/ 0,

/\* 064 4 \*/ 0,

/\* 065 5 \*/ 0,

/\* 066 6 \*/ 0,

/\* 067 7 \*/ 0,

/\* 070 8 \*/ 0,

/\* 071 9 \*/ 0,

/\* 072 : \*/ 0,

/\* 073 ; \*/ 0,

/\* 074 < \*/ 0,

/\* 075 = \*/ 0,

/\* 076 > \*/ 0,

/\* 077 ? \*/ 0,

/\* 100 @ \*/ 0,

/\* 101 A \*/ ALPHA,

/\* 102 B \*/ ALPHA,

/\* 103 C \*/ ALPHA,

/\* 104 D \*/ ALPHA,

/\* 105 E \*/ ALPHA,

/\* 106 F \*/ ALPHA,

/\* 107 G \*/ ALPHA,

/\* 110 H \*/ ALPHA,

/\* 111 I \*/ ALPHA,

/\* 112 J \*/ ALPHA,

/\* 113 K \*/ ALPHA,

/\* 114 L \*/ ALPHA,

/\* 115 M \*/ ALPHA,

/\* 116 N \*/ ALPHA,

/\* 117 O \*/ ALPHA,

/\* 120 P \*/ ALPHA,

/\* 121 Q \*/ ALPHA,

/\* 122 R \*/ ALPHA,

/\* 123 S \*/ ALPHA,

/\* 124 T \*/ ALPHA,

/\* 125 U \*/ ALPHA,

/\* 126 V \*/ ALPHA,

/\* 127 W \*/ ALPHA,

/\* 130 X \*/ ALPHA,

/\* 131 Y \*/ ALPHA,

/\* 132 Z \*/ ALPHA,

/\* 133 [ \*/ 0,

/\* 134 \ \*/ 0,

/\* 135 ] \*/ 0,

/\* 136 ^ \*/ 0,

/\* 137 \_ \*/ ALPHA,

/\* 140 ` \*/ 0,

/\* 141 a \*/ ALPHA,

/\* 142 b \*/ ALPHA,

/\* 143 c \*/ ALPHA,

/\* 144 d \*/ ALPHA,

/\* 145 e \*/ ALPHA,

/\* 146 f \*/ ALPHA,

/\* 147 g \*/ ALPHA,

/\* 150 h \*/ ALPHA,

/\* 151 i \*/ ALPHA,

/\* 152 j \*/ ALPHA,

/\* 153 k \*/ ALPHA,

/\* 154 l \*/ ALPHA,

/\* 155 m \*/ ALPHA,

/\* 156 n \*/ ALPHA,

/\* 157 o \*/ ALPHA,

/\* 160 p \*/ ALPHA,

/\* 161 q \*/ ALPHA,

/\* 162 r \*/ ALPHA,

/\* 163 s \*/ ALPHA,

/\* 164 t \*/ ALPHA,

/\* 165 u \*/ ALPHA,

/\* 166 v \*/ ALPHA,

/\* 167 w \*/ ALPHA,

/\* 170 x \*/ ALPHA,

/\* 171 y \*/ ALPHA,

/\* 172 z \*/ ALPHA,

};

/\* Function Prototypes \*/

int GetNextToken(void);

/\* Global Variables \*/

char g\_prog[4096 + 1]; /\* array in which source code resides \*/

char \*g\_cp; /\* current pointer for the lexer \*/

char \*g\_limit; /\* points to the end of the code \*/

int g\_lineNo;

char g\_token[MAX\_TOKEN];

char \*g\_tokenTypes[] = { "", "Identifier", "Operatör Plus", "Operator Minus", "Operetor Multiply", "Operator Divide", "Operator Left Paranthesis", "Operator Right Paranthesis", "Operator Assignment", "Delimiter Semicolon"};

/\* Function Definitions \*/

int main(int argc, char \*argv[])

{

FILE \*f;

int tokenId;

size\_t n;

if (argc != 2) {

fprintf(stderr, "wrong number of arguments!..\n");

exit(EXIT\_FAILURE);

}

if ((f = fopen(argv[1], "r")) == NULL) {

fprintf(stderr, "cannot open file!...\n");

exit(EXIT\_FAILURE);

}

n = fread(g\_prog, 1, 4096, f);

if (ferror(f)) {

fprintf(stderr, "cannot read file!..\n");

exit(EXIT\_FAILURE);

}

fclose(f);

g\_prog[n] = '\n';

g\_limit = &g\_prog[n];

g\_cp = g\_prog;

while ((tokenId = GetNextToken()) != -1)

printf("Token Type: %s, Token: %s\n", g\_tokenTypes[tokenId], g\_token);

return 0;

}

int GetNextToken(void)

{

char \*cp;

int count;

int tokType;

while ((g\_cmap[\*g\_cp] & BLANK)) {

if (g\_cmap[\*g\_cp] & NEWLINE) {

if (g\_cp == g\_limit)

return -1;

++g\_lineNo;

}

++g\_cp;

}

if (g\_cmap[\*g\_cp] & ALPHA) {

cp = g\_cp + 1;

while (g\_cmap[\*cp] & ALPHA)

++cp;

count = cp - g\_cp;

memcpy(g\_token, g\_cp, count);

g\_token[count] = '\0';

g\_cp = cp;

return TOKEN\_IDENTIFIER;

}

switch (\*g\_cp) {

case '+':

tokType = TOKEN\_OPERATOR\_PLUS;

break;

case '-':

tokType = TOKEN\_OPERATOR\_MINUS;

break;

case '\*':

tokType = TOKEN\_OPERATOR\_MULTIPLY;

break;

case '/':

tokType = TOKEN\_OPERATOR\_DIVIDE;

break;

case '(':

tokType = TOKEN\_LEFT\_PARANTHESIS;

break;

case ')':

tokType = TOKEN\_RIGHT\_PARANTHESIS;

break;

case ';':

tokType = TOKEN\_SEMICOLON;

break;

case '=':

tokType = TOKEN\_ASSIGNMENT;

break;

}

g\_token[0] = \*g\_cp++;

g\_token[1] = '\0';

return tokType;

}

Buradaki kodda birkaç noktanın altını çizmek istiyoruz:

- Her ne kadar kullanılmamış olsa da yukarıdaki kodda satır sayılarının tutulduğunu görüyorsunuz. Pek çok durumda (hata mesajlarında vs.) o andaki satır numarasının tutulması gerekebilmektedir.

- Tek karakterli atomların türleri için doğrudan onların ASCII karakter karşılıkları kullanılabilir. (Örneğin ‘;’ atomunun türü ‘;’ karakterinin ASCII karşılığı olarak alınabilir.) Böylece yukarıdaki kodun son bölümü çok daha basit hale gelecektir.

- Kodda sürekli “dosyanın sonuna gelindi mi?” kontrolünün yapılmadığına bunun yalnızca ‘\n’ karakteri görüldüğünde yapıldığına dikkat ediniz.

Bir C programının atomlarına ayrılması yukarıdaki örnekten daha karmaşıktır. Ancak temel yapı aynıdır. Kursumuzda Src dizinini altında “008-C-Lexer” dizininde bu işlemi yapan kodlar verilmiştir. Bu kodlarda yukarıda ele alınan tamponlama da uygulanmış durumdadır.

**3.4.3. Düzenli İfadeler (Regular Expressions)**

Düzenli ifadeleri "yazısal kalıpların ifade edilmesinde kullanılan küçük bir dil" olarak tanımlayabiliriz. Gerçekten de özellikle yazılarda belli kalıpların aranması sürecinde düzenli ifadelerden sıkça faydalanılmaktadır. Örneğin bir text editörde “dd/mm/yyyy” kalıbına uyan tarihleri ya da xxxxx@yyyyy.com kalıbına uyan e-posta adreslerini bulmak isteyebiliriz. Bu kalıpların editörlerdeki klasik metin arama özellikleriyle bulunamayacağına dikkat ediniz. İşte düzenli ifadeler böyle kalıpların ifade edilmesini sağlayan kurallar topluluğundan oluşmaktadır. Gelişmiş pek çok kelime işlemci düzenli ifadeler yoluyla arama işlemi yapabilmektedir. Lexical analiz araçlarında da atomların belirlenmesi sürecinde düzenli ifadelerden faydalanılmaktadır.

Düzenli ifadeler üzerinde işlem yapan araçların kullandıkları kodlara “düzenli ifade motorları (regular exprtession engines)” denilmektedir. Malesef düzenli ifadelerin kurallarına ilişkin bir standart yoktur. Bu nedenle düzenli ifade motorlarının da tamamen aynı kurallara sahip olduklarını söyleyemeyiz. Ancak pek çok motor büyük ölçüde birbirlerine benzemektedir.

Düzenli ifadeler yukarıda da belirtildiği gibi kalıpların ifade edilmesinde kullanılmaktadır. Düzenli ifadeleri kullanan tipik araçlardan bazıları şunlardır:

- Gelişmiş kelime işlemciler. (Microsoft Word, Libre Office vs.)

- Bazı komut satırı araçları. Örneğin bunların en ünlüsü “grep”tir.

- awk, sed gibi text işlemlerinde kullanılan küçük diller.

- Programlama dillerindeki kütüphane fonksiyonları ve sınıflar. Örneğin POSIX'in regex fonksiyonları ya da boost’tan alınarak C++11’e dahil edilmiş olan regex sınıfları.

- Lexical analiz işlemlerini yapan araçlar.

**3.4.3.1. Düzenli İfadelerin Oluşturulması**

Düzenli ifadelerde iki tür karakter kümesi vardır: Normal karakterler ve meta karakterler. Normal karakterler kalıpta karakter olarak bulunması gereken öğelerdir. Yani kalıptaki normal bir karakter başak bir şeyi değil kendisini temsil eder. Meta karakterler ise kalıpta kendisini temsil etmeyen, özel anlama gelen karakterlerdir. Örneğin '+' bir meta karakterdir. '+' karakterinin düzenli ifadelerde özel başka bir anlamı vardır. Bu karakter onun solundaki karakterden “bir tane ya da daha fazla bulunma” durumunu belirtir. Örneğin “ab+c” kalıbı aşağıdaki yazılarla uyuşabilir:

abc

abbbbc

abbbbbbbbc

abbbbbbbbbbbbc

İşte '+' gibi değişik anlamlara gelen pek çok meta karakter bulunmaktadır. Zaten düzenli ifade dilinin öğrenilmesi büyük ölçüde bu meta karakterlerin öğrenilmesi sürecidir.

Düzenli ifade motorlarının kullandığı tipik meta karakterler ve anlamları şunlardır:

|  |  |
| --- | --- |
| **Meta Karakterler** | **Anlamı** |
| . (nokta) | ‘\n’ dışındaki herhangi bir karakter |
| ? | Solundaki karakterden 0 tane ya da 1 tane |
| \* | Solundaki karakterden 0 tane ya da çok tane |
| + | Solundaki karakterden 1 tane ya da çok tane |
| {n} | Burada n bir sayıdır. Solundaki karakterden tam olarak n tane anlamına helir. |
| {n,} | Burada n bir sayıdır. Solundaki karakterden tam olarak en az n tane anlamına helir. |
| {n,m} | Burada n ve m birer sayıdır. Solundaki karakterden en az n tane en fazla m tane anlamına gelir. |
| [ ] | Köşeli parantez içerisindeki karakterlerden herhangi birisi anlamına gelir. |
| [x-y] | x ve y aralığındaki herhangi bir karakter |
| [^ ] | Köşeli parantez içerisindeki karakterlerden olmayan herhangi bir karakter |
| \w | Herhangi bir alfanümerik karakter |
| \W | Herhangi bir alfanümerik olmayan karakter |
| \s | Herhangi bir boşluk karakteri |
| \S | Herhangi bir boşluk olmayan karakter |
| $ | Satırın sonunun belli karakterlerle sonlanması durumu (Örneğin “kaan$”) |
| ^ | Satırın başı belli karakterlerle sonlanması durumu (Örneğin ^kaan”) |
| (..) | Gruplama amacıyla kullanılır. Böylece bunun sağındaki metakarakterler bu grup için anlam kazanır. |
| | | Veya anlamına gelmektedir. Örneğin "ali|veli" yazı içerisindeki "ali" veya "veli" ile uyşur. |

Düzenli ifadelerde kullanılan tüm meta karakterlerin bunlarla sınırlı olmadığını belirtelim. Diğer meta karakterler için kurs dokümanları içerisindeki kitaplardan faydalanabilirsiniz.

Parantezlerin gruplama amacıyla kullanıldığında dikkat ediniz. Örneğin ([a-z]\_){3} kalıbında {3} ‘a’ ile ‘z’ arasındaki karakterlerden biri ile ‘\_’ karakterinin birleşimlerinde üç tane olacağı anlamına gelmektedir (örneğin “x\_y\_z\_” gibi).

Kalıp içerisindeki meta karakterlerin normal karakterlerle karışmaması için düzenli ifade motorları iki yöntem kullanabilmektedir:

1) Meta karakterlerle çakışan normal karakterlerin önüne ters bölü karakteri getirme yöntemi. Örneğin: "\.+" kalıbı bir ya da birden fazla '.' karakteri ile uyuşur. Ters bölüden dolayı artık kalıptaki '.' karakteri bir meta karakter olarak değil '.' karakterinin kendisi olarak ele alınır.

2) Meta karakterlerlerin önüne ters bölü karakteri getirme yöntemi. Örneğin bu yöntemde ".\+" kalıbı bir ya da birden fazla '.' karakteriye uyuşacaktır. Bu yöntemde çakışan karakterlerin default olarak normal karakter kabul edildiğine dikkat ediniz.

Düzenli arama motorları genellikle birinci yöntemi kullanmaktadır. Ancak bazıları kullanıcının her iki yöntemden birini seçmesine de olanak sağlar.

Şimdi bu meta karakterlerin anlamlarına ilişkin bazı örnekler verelim:

|  |  |
| --- | --- |
| **Kalıp** | **Neyle Uyuşur?** |
| [\_a-zA-z]+ | '\_' karakterinden ve alfabetik karakterlerden oluşan karakter dizileriyle uyuşur. Köşeli parantez içerisindeki [a-z] gibi bir kalıbın 'a' ile 'z' arasındaki herhangi bir karakter anlamına geldiğini anımsayınız. |
| [+-]?[0-9]+\.?[0-9]\* | Gerçek sayı kalıplarıyla uyuşur. Örneğin “123”, “123.45”, “-1” gibi. (Ancak “.12” ya da “.12” gibi kalıplarla uyuşmaz) |
| ([0-9]{1,3}\.){3}[0-9]{1,3} | Bölümleri “.” ile ayrılmış IP numaralarıyla uyuşur. Örneğin “192.160.0.100” gibi. Burada parantezler gruplama amacıyla kullanılmıştır. Dolayısıyla kalıbın [0-9]{1,3} kısmı 0’dan 9’a kadar karakterlerden 1 ya da 2 ya da 3 tane olacağını belirtir. kalıbın ([0-9]{1,3}\.){3} kısmı ise üç basamağa kadar sayı ve noktalaran toplamda üç tane bulunacağını belişrtmektedir. |
| ^\w\* | Satırların başındaki sözcüklerle uyuşur |

**3.4.3.2. UNIX/Linux Sistemlerindeki grep Programı**

grep (globally search a regular **e**xpression) UNIX/Linux sistemlerinde çok sık kullanılan POSIX standartlarında da tanımlı olan bir komut satırı aracıdır. grep programının genel kullanım biçimi şöyledir:

grep <kalıp> [dosya yol ifadeleri]

Kalıp eğer boşluk içermiyorsa tırnak içerisine alınmayabilir. Ancak boşluk içeriyorsa tırnaklanmalıdır. Tırnaklama tek tırnak ya da çift tırnak ile yapılabilir. Komutta bir ya da birden fazla "dosya yol ifadesi" belirtilebilir. Bu durumda grep kalıbı sırasıyla bu dosyalarda arar. Eğer komutta hiç "dosya yol ifadesi" belirtilmezse grep arama yapacağı yazıyı stdin dosyasından okur. Örneğin:

grep -E “([0-9]{1,3}\.){3}[0-9]{1,3}” test.txt

Burada grep kalıbı "test.txt" dosyasında arayacaktır.

grep default durumda metin içerisinde kalıbı bulduğunda onun bulunduğu satırın tamamını ekrana (stdout dosyasına) yazdırmaktadır. Kalıp içerisinde meta karakterler ile çakışan normal karakterlerin hangilerinin ters bölü ile yazılacağı komut satırından –E seçeneği ile belirlenir. grep default durumda kalıptaki her karakteri normal karakter olarak değerlendirir. Bunların meta karakter olarak değerlendirilmesi için ters bölülenmesi gerekir. Halbuki –E seçeneği bunun tam tersine yol açmaktadır. –E seçeneği girilirse kalıptaki meta karakterlerle çakışan karakterler meta karakterler kabul edilir. Bu durumda bunları meta karakter olmaktan çıkartmak için ters bölülemek gerekecektir. grep'in "–E" kullanımı çok yaygın olduğu için UNIX/Linux sistemlerinde bu işlemi yapan "egrep" komutu da bulundurulmaktadır. Yani "egrep" ile "grep -E" aynı işlemeyol açmaktadır.

grep borulama işlemleriyle de çok sık kullanılmaktadır. Örneğin:

ps -e | grep "tty"

Burada borulama sayesinde ps komutunun stdout dosyasına yazdıklarını grep stdin’den okuyormuş gibi bir etki oluşacaktır. Sonuç olarak bu komutla birlikte proses listesindeki içinde "tty" geçen satırlar elde edilecektir. Örneğin:

ls -l /usr/include | grep "Nov"

Burada /usr/iclude dizininde “Nov” geçen satırlar listelenmektedir.

Yukarıda da belirtildiği gibi grep default olarak kalıbın bulunduğu tüm satırı ekrana (stdout dosyasına) yazdırmaktadır. Fakat eğer grep “–o” seçeneğiyle kullanılırsa kalıbın bulunduğu tüm satırı değil yalnızca bulunan kalıbın kendisini ekrana yazdırır. “–c” seçeneği kalıbın toplamda kaç satırda bulunduğu bilgisini, “-b” seçeneği kalıbın dosyanın kaçıncı offset’lerinde bulunduğu bilgisini, “-n” seçeneği de kalıbın bulunduğu satır numaralarını vermektedir.

**3.4.3.3. C’de Düzenli İfadelerle İşlemler**

C’nin standart kütüphanesinde düzenli ifadeler üzerinde işlem yapan fonksiyonlar yoktur. Ancak POSIX standartlarında düzenli ifadeler üzerinde işlem yapan C fonksiyonları bulunmaktadır. POSIX fonksiyonlarının UNIX türevi işletim sistemlerinde bulunması öngörülmüş olan fonksiyonlar olduğunu anımsayınız. Bu nedenle düzenli ifadelere ilişkin POSIX fonksiyonlarını da yalnızca UNIX/Linux türevi sistemlerde ve Mac OS X sistemlerinde kullanabilirsiniz.

Düzenli ifadeler üzerinde işlem yapan POSIX fonksiyonları şunlardır:

#include <sys/types.h>

#include <regex.h>

int regcomp(regex\_t \*preg, const char \*regex, int cflags);

int regexec(const regex\_t \*preg, const char \*string,

size\_t nmatch, regmatch\_t pmatch[], int eflags);

size\_t regerror(int errcode, const regex\_t \*preg, char \*errbuf, size\_t errbuf\_size);

void regfree(regex\_t \*preg);

regcomp fonksiyonu düzenli ifadeyi oluşturan ana fonksiyondur. Bu fonksiyondan regext\_t türüyle temsil edilen bir handle değeri elde edilmektedir. regcomp fonksiyonun birinci parametresi regex\_t türünden içi doldurulacak nesnenin adresini alır. Fonksiyonun ikinci parametresi düzenli ifade kalıbını almaktadır. Son parametre ise bazı seçeneklerden oluşmaktadır. Fonksiyon başarı durumunda sıfır, başarısızlık durumunda hata kodunun kendisine geri dönmektedir.

Kalıbı bulan asıl fonksiyon regexec fonksiyonudur. Bu fonksiyonun birinci parametresi regcomp fonksiyonundan elde edilen handle değerini, ikinci parametresi de arama yazısını almaktadır. Üçücüncü parametre eşleşen kaç kalıbın ve alt kalıpların sayısını belirtir. Bulunan kalıplar regmatch\_t türünden bir dizinin içerisine yerleştirilmektedir. Son parametre yine bazı seçenekleri belirtmektedir.

Düzenli ifadelerle işlemler bittiğinde regcomp fonksiyonunda yapılan bazı tahsisatları geri almak için regfree fonksiyonu çağrılmalıdır. Örneğin:

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <regex.h>

#define BUF\_SIZE 10000

int main(int argc, char \*argv[])

{

regex\_t rex;

FILE \*f;

size\_t n;

char buf[BUF\_SIZE + 1];

regmatch\_t matches[1];

int result, i, k, beg = 0;

if (argc < 2 || argc > 3) {

fprintf(stderr, "wrong number of arguments!\nusage mygrep <pattern> [path]\n");

exit(EXIT\_FAILURE);

}

if (argc == 2)

f = stdin;

else {

if ((f = fopen(argv[2], "r")) == NULL) {

fprintf(stderr, "cannot open file!..\n");

exit(EXIT\_FAILURE);

}

}

n = fread(buf, 1, BUF\_SIZE, f);

if (n == 0 && ferror(f)) {

fprintf(stderr, "cannot read file!..\n");

exit(EXIT\_FAILURE);

}

buf[n] = '\0';

if ((result = regcomp(&rex, argv[1], REG\_EXTENDED)) != 0) {

fprintf(stderr, "%s\n", strerror(result));

exit(EXIT\_FAILURE);

}

while (regexec(&rex, buf + beg, 1, matches, REG\_NOTEOL) != REG\_NOMATCH) {

printf("%.\*s\n", matches[0].rm\_eo - matches[0].rm\_so, &buf[beg + matches[0].rm\_so]);

beg += matches[0].rm\_eo + 1;

}

regfree(&rex);

return 0;

}

Biz burada yalnızca temel bazı açıklamalar eşliğinde küçük bir örnek verdik. Ancak POSIX'in düzenli ifadeler üzerinde işlem yapan fonksiyonlarının bazı ayrıntıları da vardır. Bu ayrıntılar hakkında bilgileri POSIX dokümanlarından ya da "man" sayfalarından elde edebilirsiniz.

**3.4.3.4. C++’ta Düzenli Düzenli İfadelerle İşlemler**

C++’ta düzenli ifadeler ile işlemler için ağırlıklı olarak "boost" kütüphanesindeki regex sınıfları kullanılıyordu. Bu sınıflar daha sonraları üzerinde küçük değişiklikler yapılarak C++11’de standartlara dahil edilidi. Bu kurs C++ bilgisi gerektirmediği için biz burada yalnızca C++11'in regex sınıflarına yönelik küçük bir örnekle yetineceğiz.

#include <iostream>

#include <regex>

using namespace std;

int main()

{

regex re("(\\+|-)?[0-9]+");

string text;

cout << "Bir yazi giriniz:";

cin >> text;

if (regex\_match(text, re))

cout << "yazida bir sayi var\n";

else

cout << "yazida sayi yok!\n";

return 0;

}

Burada bir regex nesnesi oluşturulmuştur. Bu nesne oluşturulurken düzenli ifade kalıbı sınıfın başlangıç fonksiyonunda (constructor) belirtilmiştir. Sonra regex\_match fonksiyonu ile verilen yazının bu kalıba uyup uymadığına bakılmıştır. Yazı içerisindeki kalıbın aranması işlemi ise regex\_search fonksiyonuyla yapılmaktadır. Örneğin:

#include <iostream>

#include <regex>

using namespace std;

int main()

{

regex re("(\\+|-)?[0-9]+");

string text;

smatch match;

cout << "Bir yazi giriniz:";

cin >> text;

if (regex\_search(text, match, re)) {

cout << "prefix: " << match.prefix() << ", suffix: " <<

match.suffix() << endl;

cout << "bulunan kalip:";

for (int i = 0; i < match.size(); ++i)

cout << match[i];

cout << endl;

}

else

cout << "not ok\n";

return 0;

}

C++11’in regex kütüphanesinin ayrıntıları için ilgili dokümanları inceleyiniz.

**3.5. Lexical Analiz İşlemini Yapan Araçlar**

Lexical analiz işlemleri manuel olarak yapılabileceği gibi bazı araçlar kullanılarak da yapılabilmektedir. Lexical analiz işlemini gerçekleştiren pek çok araç bulunmaktadır. Ancak bunlardan en yaygın kullanılanının “lex” ve onun modern versiyonu olan “flex” olduğu söylenebilir. ANTLR diğer bir seçenek olarak düşünülebilir. ANTLR gittikçe yaygınlaşmaktadır.

Lexical analiz araçları lexical analiz işlemini yapan program kodlarını üretmektedir. Programcı da üretilen bu kodları kendi projelerine ekleyerek kullanabilmektedir. lex 1975 yılında tasarlanmış olan standart haline gelmiş bir lexical analiz aracıdır. Flex ise 1987 yılında lex’in geliştirilmiş bir biçimi olarak tasarlanmıştır. flex ile lex pek çok bakımından uyumludur. Flex organik bağlantı olsa da resmi olarak GNU projesi kapsamında değildir.

**3.5.1. Flex (Lex) Aracının Kullanımı**

Flex orijinal olarak UNIX/Linux sistemleri için düşünülmüşür. Ancak bu aracın Windows versiyonu da oluşturulmuş durumdadır. Flex Linux sistemlerinde “binutil” paketi içersinde işletim sisteminin temel bir aracı olarak bulundurulmaktadır. Yani Linux sistemlerinde flex’in kurulumu zaten yapılmış gibidir. Ancak Flex'in Windows sürümleri numara olarak daha geriden gelmektedir. Flex’in orijinal dokümanları kendi sitesinden indirilebilir. Bunları kurs dokümanları içerisinde “Doc/EBooks/Flex-Bison” dizininde bulabilirsiniz.

**3.5.1.1. Flex’in Kurulumu**

Yukarıda da belirtildiği gibi Flex Linux sistemlerinin temel bir aracı olarak kabul edilmektedir. Eğer sistemde gcc derleyicisi yüklü ise (temel kurulumlarda bile gcc yüklenir) Flex de zaten yüklenmiş durumda olacaktır. Tabii biz bu sistemlerde kurulum programlarıyla bunların yeni versiyonlarını indirip kurabiliriz. Örneğin:

sudo apt-get install flex

Yukarıda da belirttiğimiz gibi Flex’in Windows versiyonu da oluşturulmuştur. Projenin “sourceforge.net” sayfasının adresi şöyledir: <http://gnuwin32.sourceforge.net/packages/flex.htm>. Bu sayfadan kurulum dosyası indirilerek kurulum yapılabilir. Tabii Flex komut satırından kullanılan bir araç olduğu için kurulumdan sonra “PATH” çevre değişkeninin ayarlanması gerekmektedir. Ancak Flex’in bu Windows versiyonunda uzun komut satırı seçenekleri konusunda (örneğin –header-file gibi) sorunlar vardır. Buna alternatif olarak <https://sourceforge.net/projects/winflexbison/> projesindeki binary dosyalar da kullanılabilir. Bu projedeki Flex programının ismi “win\_flex.exe” biçimindedir.

Mac OS X sistemlerinde çalışıyorsanız Flex'i “brew” utility’si yardımıyla kurabilirsiniz. Tabii önce “brew” utility’sini kurmair. Bu işlem şöyle yapılabilir:

ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)" < /dev/null 2> /dev/null

Daha sonra "brew" ilw kurulum şöye yapılabilir:

brew install flex

**3.5.1.2. Flex’in Genel Sentaksı**

Bir Flex kaynak dosyası üç bölümden oluşmaktadır: “Tanımlamalar (Definitions)”, Kurallar (Rules)” ve “Kullanıcı Kodları (Uer Codes)”. Kaynak kodun tepesinden ilk %% karakterlerine kadarki bölüm “tanımlamaları”, ilk %% karakterlerinden ikinci %% karakterlerine kadar olan bölüm “kuralları” ve ikinci %% karakterlerinden dosya sonuna kadarki bölüm de “kullanıcı kodlarını” oluşturmaktadır:

Tanımlamalar (definitions)

%%

Kurallar (Rules)

%%

Kullanıcı Kodları (User Codes)

Flex kodları herhangi bir text editör kullanılarak oluşturulabilir. Geleneksel olarak Flex dosyalarının uzantısı “.l” biçiminde verilmektedir. Tabii aslında uzantı herhangi bir biçimde olabilir.

Flex dosyası oluşturulduktan sonra dosya işlenmek üzere komut satırından flex programına sokulur. Bu işlemin en yalın hali şöyledir:

flex <flex kaynak dosyası>

Örneğin:

flex sample.l

Flex bu işlem sonucunda bize lexical analiz işlemini yapan bir C dosyası verecektir. Flex'in ürettiği bu C dosyası default olarak “lex.yy.c” ismindedir. Artık biz bu dosyayı bir C derleyicisi ile derleyerek lexical analiz işlemini yapan programımızı çalışabilir hale getirebiliriz. İleride de ele alacağımız gibi flex bize default biçimde yazılmış iki fonksiyonu bir kütüphane içerisinde vermektedir. Bunlardan birincisi programın başlangıç noktası olan main fonksiyonu (yanlış görmüyorsunuz main fonksiyonu da istenirse kütüphaneye yerleştirilebilir) diğeri de yywrap isimli fonksiyondur. Bu iki fonksiyon “libfl.a” isimli kütüphanedir. Eğer main ve yywrap fonksiyonalrını biz yazmayacaksak link işlemine bu kütüphanenin de dahil edilmesi gerekir. Bu dahil etme işlemini gcc derleyicilerinde “-lfl” seçeneği ile yapabilirsiniz. Örneğin:

gcc –o sample lex.yy.c –lfl

Windows’ta gcc derleyicisiyle derleme yapılırken eğer “-lfl” seçeneği kullanılacaksa libfl.a’nın yeri önemlidir. Bu yerin –L seçeneğiyle belirtilmesi gerekir. Örneğin:

gcc -o sample lex.yy.c -LC:\Program Files (x86)\GnuWin32\lib -lfl

Ya da Windows’ta hiç “-lfl” seçeneği kullanılmadan doğrudan “.a” uzantılı kütüphane dosyasının yeri de belirtilebilir. Örneğin:

gcc –o sample lex.yy.c “C:\Program Files (x86)\GnuWin32\lib\libfl.a”

Windows’ta Microsoft’un “cl.exe” derleyicisi ile derleme yapılırken bu “libfl.a” kütüphanesi benzer biçimde link işlemine dahil edilmelidir.

cl /output: sampl.exe lex.yy.c “C:\Program Files (x86)\GnuWin32\lib\libfl.a”

Burada bir noktaya dikkatinizi çekmek istiyoruz. Eğer main fonksiyonunu programcının kendisi yazarsa (zaten biz hep böyle yapacağız) ve yywrap fonksiyonunu da yazmayacağını beyan ederse "libfl.a" dosyasının link işlemine dahil edilmesinin gerekliliği ortadan kalkmaktadır. yywrap fonksiyonunun bulunmayacağı bilgisi flex kaynak dosyasında "%option noyywrap" direktifi ile belirtilmektedir. Bu durumda iskelet bir flex programı şöyle olabilir:

%option noyywrap

%%

%%

int main(void)

{

yylex();

return 0;

}

Artık derleme işleminde “libfl.a” dosyasının belirtilmesine gerek yoktur. Hem Linux, hem Mac OS X hem de Windows sistemlerinde flex ve derleme işlemleri şöyle yapılabilir:

flex sample.l

gcc –o sample lex.yy.c

Windows’ta Microsoft’un “cl.exe” derleyicisi ile derleme işlemini şöyle yapabilirsiniz:

cl /output:sample.exe lex.yy.c

Yukarıda da belirttiğimiz gibi default olarak flex programının ürettiği C program dosyası “lex.yy.c” ismindedir. Ancak biz "–o" komut satırı argümanıyla üretilecek dosyanın ismini belirleyebiliriz. Örneğin:

flex –osample.c sample.l

gcc –o sample sample.c

flex klasik GNU argüman seçenekleri kurallarına uymamaktadır. Bu nedenle –o seçeneği ile dosya ismi arasında boşluk karakterleri bırakmaynız.

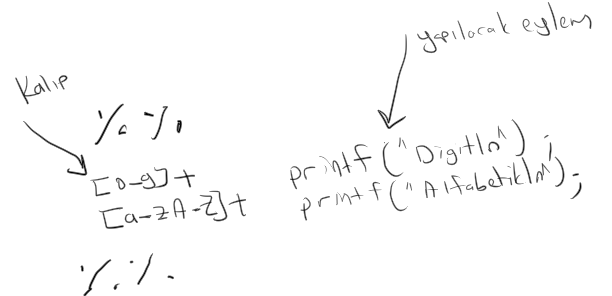
**3.5.1.3. Flex Kaynak Dosyasındaki Bölümlerin Anlamı**

Flex aracının ürettiği koddaki lexical analiz işlemini yapan temel fonksiyon yylex isimli fonksiyondur. Bu fonksiyon bizim önceki konularda ele aldığımız GetNextToken fonksiyonuna benzetilebilir. yylex tek çağırmada her şeyi yapacak biçimde ya da her çağırmada sıradaki atomu verecek biçimde de kullanılabilir. İskelet Flex programında main fonksiyonu içerisinde yylex fonksiyonun yalnızca bir kez çağrıldığına dikkat ediniz.

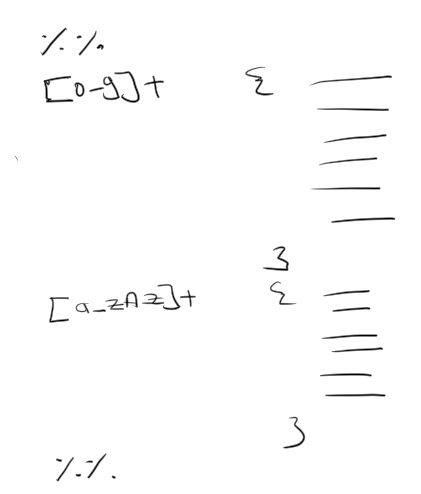
Flex kaynak dosyasndaki en önemli bölüm “Kurallar (Rules)” bölümüdür. Kurallar bölümünün iki %% karakterlerinin arasındaki bölüm olduğunu anımsayınız. Bu bölüm “kalıp (pattern)” ve “yapılacak eylem (action)” çiftlerinden oluşmaktadır:

kalıp (pattern) yapılacak eylem (action)

Kalıp en soldaki sütuna dayalı olarak (unindented) biçimde yazılmak zorundadır. (Yani kalıpları yazarken soldan boşluk karakterleri bile veremeyiz.) Kalıp düzenli ifadelerden oluşur. Kalıptan sonraki boşluk karakterleri atılarak ilk boşluksuz karakterden satır sonuna kadarki karakterler “yapılacak eylemi (action)” olarak belirlenmektedir. Örneğin:



Yapılacak eylem C kodlarından oluşmaktadır. Başka bir deyişle “yapılacak eylem (action)” ilgili kalıp bulunduğunda çalıştırılacak C kodlarını belirtir. Eğer “yapılacak eylem” bir satırdan fazlaysa blok açılarak kodlar istenildiği kadar uzatılabilir. Örneğin:



Aslında bir kalıp için bir “yapılacak eylem” kısmının bulunması da zorunlu değildir. eğer kalıp için bir "yapılacak eylem" belirtilmediyse o kalıp bulunduğunda hiçbir şey yapılmaz.

Flex’in ürettiği kod (“lex.yy.c” dosyasındaki kod) default durumda lexical analiz işlemine tutulacak kaynak metni "stdin" dosyasından (yani klavyeden) okumaktadır. Eğer programcı girişlerin "stdin" dosyasından değil de kendi istediği bir dosyadan alınması istiyorsa ilgili dosyayı fopen fonksiyonyla açıp dosya bilgi göstericisini de Flex'in global yyin isimli FILE \* türünden değişkenine atamalıdır. Örneğin:

%option noyywrap

%{

#include <stdio.h>

#include <stdlib.h>

%}

%%

[0-9]+ {

printf("digit\n");

}

\n

.

%%

int main(void)

{

if ((yyin = fopen("test.txt", "r")) == NULL) {

fprintf(stderr, "cannot open file!\n");

exit(EXIT\_FAILURE);

}

yylex();

return 0;

}

Burada artık "test.txt" dosyası lexical analiz işlemine sokulacaktır. Kodu aşağıdaki gibi derleyebilirsiniz:

flex –osample.c sample.l

gcc –o sample sample.c (ya da Microsoft derleyicilerinde “cl sample.c”)

Burada “tanımlamalar (definitions)” bölümündeki %{ ve %} kısmında iki include işleminin yapıldığını görüyorsunuz. Tanımlamalar bölümünü biraz ileride ele alacağız. yyin değişkeni default olarak “stdin” dosyasına ilişkin dosya bilgi göstericisini tutmaktadır. Biz örneğimizde yyin değişkenine “test.txt” dosyasına ilişkin dosya bilgi göstericisini atayarak yylex fonksiyonunun bu dosyadan okuma yapmasını sağladık. Şimdi de aşağıdaki örneği inceleyiniz:

%option noyywrap

%{

#include <stdio.h>

#include <stdlib.h>

%}

%%

[0-9]+ {

printf("digit\n");

}

\n

.

%%

int main(int argc, char \*argv[])

{

if (argc > 2 ) {

fprintf(stderr, "wrong number of arguments!\n");

exit(EXIT\_FAILURE);

}

if (argc == 2)

if ((yyin = fopen(argv[1], "r")) == NULL) {

fprintf(stderr, "cannot open file!\n");

exit(EXIT\_FAILURE);

}

yylex();

return 0;

}

Kodu aşağıdaki gibi derleyebilirsiniz:

flex –osample.c sample.l

gcc –o sample sample.c (ya da Microsoft derleyicilerinde “cl sample.c”)

Bu örnekte komut satırı argümanıyla verilen dosya fopen fonksiyonuyla açılarak dosyaya ilişkin dosya bilgi göstericisi yyin değişkenine atanmıştır. Böylece programın girdilerinin komut satırı argümanı ile verilen dosyadan okunması sağlanmıştır. Programı aşağıdaki gibi komut satırı argümanı vererek çalıştırmalısınız:

sample test.txt (Windows)

./sample test.txt (Linux/Mac OS X)

Flex default durumda kalıba uymayan bütün karakterleri “stdout” dosyasına yazdırmaktadır. Bu durumda örneğin aşağıdaki flex programı dosya içerisindeki “ankara” yazılarını diğer karakterleri değiştirmeden “ANKARA” olarak ekrana yazdıracaktır:

%option noyywrap

%{

#include <stdio.h>

#include <stdlib.h>

%}

%%

ankara printf("ANKARA");

%%

int main(int argc, char \*argv[])

{

if (argc > 2 ) {

fprintf(stderr, "wrong number of arguments!\n");

exit(EXIT\_FAILURE);

}

if (argc == 2)

if ((yyin = fopen(argv[1], "r")) == NULL) {

fprintf(stderr, "cannot open file!\n");

exit(EXIT\_FAILURE);

}

yylex();

return 0;

}

Eğer kalıba uymayan karakterlerin “stdout” dosyasına yazdırılmasını istemiyorsak “kurallar” kısmının sonuna “yapılacak eylem (action)” kısmı olmayan “.” ve “\n” kalıplarını yerleştirebiliriz. (Anımsanacağı gibi düzenli ifadelerde “.” meta karakteri “\n dışındaki herhangi bir karakter” anlamına gelmektedir.) Örneğin:

%%

ankara printf("ANKARA");

\n

.

%%

Artık “ankara” kalıbına uymayan her karakter “.” ya da “\n” kalıbına uymaktadır. Bu kalıplar için birşey yapılmadığından “ankara” kalıbının dışındaki karakterlerin “stdout” dosyasına yazılması engellenmiş olur.

Flex’te “kurallar” kısmında belirtilen kalıplar için iki önemli nokta vardır:

1) Eğer yazıdaki bir kısım birden fazla kuraldaki kalıba uyuyorsa en uzun uygun kalıba ilişkin eylem işletilir. Örneğin:

%%

\+ printf(“+ operatörü\n”);

\+= printf(“+= operatörü\n”);

%%

Burada kalıptaki ‘+’ karakteri meta karakter değil normal ‘+’ karakteri olduğu için ters bölülenmiştir. Yukarıdaki kurallarda yazıda eğer tek bir + görülmüşse birinci kalıp bununla uyuşur (match eder). Fakat += görülmüşse ikinci kalıp daha uzun olduğu için ikinci kalıp uyuşur. Benzer biçimde:

%%

ali printf(“ali bulundu\n”);

aliye printf(“aliye bulundu\n”);

%%

Burada yazı içerisinde “alim” gibi bir dizilim varsa bununla birinci kalıp uyuşacaktır. Ancak “aliyem” gibi bir yazıyla ikinci kalıp uyuşacaktır.

2) Eğer yazıdaki bir kısım birden fazla kuraldaki kalıba uyuyorsa ve bunlar da aynı uzunluktaysa daha yukarıda yazılmış olan kalıbın “yapılacak eylem (action)” kısmı işletilir. Örneğin:

%%

123 printf("birinci kural\n");

[0-9]+ printf("ikinci kural\n");

.

\n

%%

Burada örneğin yazıda “123” gibi bir dizilim görülürse bu aslında eşit uzunluklu olarak birinci kalıba da ikinci kalıba da uymaktadır. Ancak Flex bu durumda daha yukarıya yazılmış kalıbın eylem kısmını işletecektir.

Kurallar bölümünün sentaksı maddeler halinde aşağıdaki gibi özetlenebilir:

1) Kurallardaki kalıplar en soldaki sütuna dayalı (unindented) yazılmalıdır.

2) Kurallardaki kalıplar içinı “yapılacak eylem (action)” kısmı hiç belirtilmeyebilir. Bu durumda bu kalıp ile uyuşma durumunda birşey yapılmayacaktır.

3) Kuralların “yapılacak eylem (action)” kısımları tek bir satırda bulunmak zorundadır. Eğer bunların birden fazla satırda yer alması isteniyorsa bloklama yapılmalıdır. Blokların içleri istenildiği gibi yazılabilir. Bu durumda kalıbın “yapılacak eylem (action)” kısmının blok kapatıldığında sonlandığı kabul edsilmektedir.

4) Kurallardaki kalıplar en soldaki sütuna dayalı olarak yazılmalıdır. Eğer kalıplar en soldaki sütuna dayalı olarak yazılmazlarsa arık bunlar kalıp olmaktan çıkmaktadır. Flex dokümanları kalıp olarak ele alınmayan bu karakterlerin sanki C koduymuş gibi ele alınacağını ve üretilen C koduna aktarılacağını belirtmektedir. Ancak dokümanlar bu karakterlerin üretilen C kodunun neresine aktarılacağı konusunda bir garanti vermemektedir. O halde bir satıra kalıbı olmayan bir kod yazmak iyi bir teknik değildir. Ancak Flex dokümanları özel bir durum olarak “kurallar kısmının” başındaki kodların yylex fonksiyonunun ana bloğunun başına aktarılacağı konusunda garanti vermektedir. (Yani biz örneğin “kurallar” kısmının başında bildirim yaparsak bu bildirilen değişkenler yylex fonksiyonunun yerel değişkenleri gibi olacaktır.)

5) Kurallar kısmında (ileride de görüleceği gibi “tanımlamalar” kısmında da) %{ ile %} arasındaki bölümler yine üretilen hedef koda aktarılmaktadır. %{ ve %} karakterleriin en soldaki sütuna dayalı olarak yazılması (unidented) gerekmektedir. Ancak bunların arasındaki kodlar istenildiği gibi yazılabilirler. Örneğin:

%%

%{

int x = 0;

%}

[0-9]+ {

if (x == 0) {

/\* ...

}

}

%%

Aslında biz yukarıdaki Flex kodunu %{ ve %} karakterlerini kullanmadan şöyle de yazabilirdik:

%%

int x = 0;

[0-9]+ {

if (x == 0) {

/\* ...

}

}

%%

Ancak %{ ... %} bloklaması ile birden fazla satıra yayılmış biçimde kodlar oluşturulabilmektedir.

Şimdi de Flex kaynak dosyasının başında bulunması gereken "tanımlamalar (definitinos)" bölümü üzerinde duralım. Tanımlamalar bölümü flex kaynak dosyasının başından ilk %% karakterlerine kadarki bölümdür. Tanımlamalar bölümünde üç şey bulunabilmektedir:

1) %option ile başlayan Flex direktifleri

2) Flex Makroları

3) Üretilecek koda yerleştirilecek global düzeydeki C kodları

%option ile başlayan satırlara Flex direktifleri denilmektedir. Flex direktifleri Flex kaynak dosyasının Flex programına bazı yönergeler vermek için kullanılmaktadır. Zorunlu olmasa da Flex direktifleri genellikle kaynak kodun tepesine yerleştirilirler. Örneğin yukarıdaki iskelet programımızda da yywrap fonksiyonunu kullanmak istemediğimizi belirtmek için kaynak kodun tepesine aşağıdaki direktifi yerleştirmiştik:

%option noyywrap

%option Fleks direktifleri en soldaki sütuna dayalı olarak (unindented) yazılmak zorundadır.

Flex makroları yazımı kolaylaştırmak ve okunabilirliği artırmak için kullanılmaktadır. Makroların genel formatı şöyledir:

isim (name) kalıp (pattern)

Makrolar da en soldaki sütuna dayalı olarak (unindented) yazılmak zorundadır. İsimden sonra boşluk karakterleri atılarak elde edilen ilk boşluksuz yazı kümesi kalıbı oluşturmaktadır. Örneğin:

Constant [\+-]?[0-9]+

Identifer [\_a-zA-Z]+

Kurallar bölümündemakrodaki isimler küme parantezlerine alınırsa bu isimler yerine ona karşı gelen kalıpların kullanıldığı kabul edilir. Örneğin:

%option noyywrap

Constant [\+-]?[0-9]+

Identifier [\_a-zA-Z]+

%%

{Constant} printf("digit\n");

{Identifier} printf("identifier\n");

.

\n

%%

Tanımlamalar kısmında sola dayalı olarak yazılmayan (indented) her şey Flex tarafından C kodu olarak ele alınmaktadır. Buradaki C kodları Flex tarafından üretilen kodun global alanına yerleştirilmektedir. Programcılar Flex'te kullanacakları global değişkenlerin ve fonksiyon prototiplerini tipik olarak tanımlamalar bölümünde bu biçimde yaparlar. Örneğin:

%option noyywrap

int g\_x; /\* Global değişken tanımlaması \*/

Constant [\+-]?[0-9]+

Identifier [\_a-zA-Z]+

%%

Yine tanımlamalar bölümünde %{ ile %} karakterleri arasındaki kısım C kodu olarak üretilen kodda global alana aktarılmaktadır. %{ ve %} karakterleri soldaki sütuna dayalı olarak (unindented) yazılmak zorundadır. Ancak bunların içi herhangi bir biçimde yazılabilir. %{ ve %} içerisine C kodlarının yazılması daha serbest ve düzenli bir görünüm sağlamaktadır. Örneğin:

%option noyywrap

%{

#include <stdio.h>

#include <stdlib.h>

int g\_x; /\* Global değişken tanımlaması \*/

%}

Constant [\+-]?[0-9]+

Identifier [\_a-zA-Z]+

%%

İstisna olarak C kodları için yorumlama başlangıcı (yani /\* karakterleri) en soldaki sütuna dayalı biçimde (unindented) yazılabilmektedir. Yorumlama kısımları da üretilen koda doğrudan aktarılmaktadır. Örneğin:

%option noyywrap

%{

#include <stdio.h>

#include <stdlib.h>

%}

/\* Global Tanımlamalar \*/

int g\_x; /\* Global değişken tanımlaması \*/

Constant [\+-]?[0-9]+

Identifier [\_a-zA-Z]+

%%

Burada /\* Global Tanımlamalar \*/ yorumlamasının en soldaki sütuna dayalı olarak yazılabildiğine dikkat ediniz. Genel olarak Flex’te her zaman C kodları için C stili yorumlama kullanılabilmektedir. Ancak makro satırlarında youmlama yapılamaz ve kurallar bölümünde de sola dayalı olarak yorumlama yapılamaz.

Flex kaynak dosyasında ikinci %% karakterlerinden sonraki bölüme "kullanıcı Kodları (user codes)" bölümü denilmektedir. Bu bölüm tamamen C kodlarına ayrılmıştır. Burada biz her türlü global tanımlamaları ve bildirimleri (örneğin fonksiyon ve global nesne tanımlamaları, tür bildirimleri gibi) yapabiliriz. Tabii bu kısımda bildirilmiş olan değişkenleri kalıplar kısmında kullanamazsınız. Eğer birtakım değişkenleri her yerde kullanmak istiyorsanız onların bildirimlerini “tanımlamalar (definitions)” bölümünde yapmalısınız. Ayrıca eğer istenirse “kullanıcı kodları (user codes)” bölümü tamamen boş da bırakılabilmektedir.

**3.5.1.4. Flex’in Önemli Global Değişkenleri**

Flex ne zaman yyin değişkeni ile belirlenen dosyada kalıba uygun bir atom bulsa o atomu yytext isimli bir değişkenin belirttiği adrese -sonunda ‘\0’ karakterini de ekleyerek- yerleştirir. yytext Flex tarafından char türden bir dizi ya da bir gösterici olarak bildirilebilmektedir. (Tabii her iki durumda da yytext adresinin gösterdiği yer tahsis edilmiş durumdadır.) Default durumda yytext değişkenini char türden bir göstericidir. Ancak bu durum %array ve direktifi ile değiştirilebilir. Aşağıdaki örneği inceleyiniz:

%option noyywrap

%{

#include <stdio.h>

#include <stdlib.h>

%}

Constant [\+-]?[0-9]+

Identifier [\_a-zA-Z]+

%%

{Constant} printf("Constant: %s\n", yytext);

{Identifier} printf("identifier: %s\n", yytext);

\n

.

%%

int main(int argc, char \*argv[])

{

if (argc > 2 ) {

fprintf(stderr, "wrong number of arguments!\n");

exit(EXIT\_FAILURE);

}

if (argc == 2)

if ((yyin = fopen(argv[1], "r")) == NULL) {

fprintf(stderr, "cannot open file!\n");

exit(EXIT\_FAILURE);

}

yylex();

return 0;

}

Kodu şöyle derleyebilirsiniz:

flex –osample.c sample.l

gcc –o sample sample.c (ya da Microsoft derleyicilerinde “cl sample.c”)

Kodun aşağıdaki gibi çalıştırıldığını varsayalım:

sample test.txt (ya da Linux ve Mac OS X'te “./sample test.txt”)

Örneğin buradaki "test.txt" dosyasının içeriği şöyle olsun:

ankara izmir 123 456789, ali, vel, selami

Programın çıktısı da şöyle olacaktır:

identifier: ankara

identifier: izmir

Constant: 123

Constant: 456789

identifier: ali

identifier: vel

identifier: selami

Flex’in int türden yyleng isimli global değişkeni, bulunan atomun karakter uzunluğunu bize verir. Yani başka bir deyişle yyleng bize strlen(yytext) değerini vermektedir. Biz yine bunu istediğimiz yerde kullanabiliriz. Örneğin:

%option noyywrap

%{

#include <stdio.h>

#include <stdlib.h>

%}

Constant [\+-]?[0-9]+

Identifier [\_a-zA-Z]+

%%

{Constant} printf("Constant: %s (%d)\n", yytext, yyleng);

{Identifier} printf("identifier: %s (%d)\n", yytext, yyleng);

\n

.

%%

int main(int argc, char \*argv[])

{

if (argc > 2 ) {

fprintf(stderr, "wrong number of arguments!\n");

exit(EXIT\_FAILURE);

}

if (argc == 2)

if ((yyin = fopen(argv[1], "r")) == NULL) {

fprintf(stderr, "cannot open file!\n");

exit(EXIT\_FAILURE);

}

yylex();

return 0;

}

Nasıl yyin atomlarına ayrılacak dosyayı belirtiyorsa, yyout da kalıp ile uyuşmayan karakterlerin yazdırıldığı dosyayı belirtmektedir. Default durumda yyout stdout dosyasına ilişkindir. Yani kalıplara uymayan karakterler default durumda stdout dosyasına yazdırılırlar.

**3.5.1.5. yylex Fonksiyonunun Atomlar İçin Tek Tek Çağrılması**

Yukarıdaki örneklerde biz main fonksiyonu içerisinde yalnızca bir kez yylex fonksiyonunu çağırdık. Bu fonksiyon başından sonuna kadar yyin ile belirtilen dosyadaki yazıyı girdi olarak kullanıyordu. Halbuki biz yylex fonksiyonunun her çağrıda sıradaki atomu vermesini sağlayabiliriz. Bunun için “kurallar (rules)” bölümünde kalıplara karşı gelen “yapılacak eylem (action)” kısımlarında yylex'in return deyimi ile sonlandırılması gerekir. Örneğin:

%option noyywrap

%{

#include <stdio.h>

#include <stdlib.h>

#define CONSTANT 1

#define IDENTIFIER 2

%}

Constant [\+-]?[0-9]+

Identifier [\_a-zA-Z]+

%%

{Constant} return CONSTANT;

{Identifier} return IDENTIFIER;

\n

.

%%

int main(int argc, char \*argv[])

{

int tokenId;

if (argc > 2 ) {

fprintf(stderr, "wrong number of arguments!\n");

exit(EXIT\_FAILURE);

}

if (argc == 2)

if ((yyin = fopen(argv[1], "r")) == NULL) {

fprintf(stderr, "cannot open file!\n");

exit(EXIT\_FAILURE);

}

while ((tokenId = yylex()) != 0) {

switch (tokenId) {

case CONSTANT:

printf("Constant: %s (%d)\n", yytext, yyleng);

break;

case IDENTIFIER:

printf("Identifier: %s (%d)\n", yytext, yyleng);

break;

}

}

return 0;

}

Flex kaynak kodunda da görüldüğü gibi yylex fonksiyonu her çağrıldığında bir atomu bulur bulmaz return deyimi ile sonlandırılmıştır. yylex dosyanın sonuna geldiğinde 0 ile geri dönmektedir. Yukarıdaki Flex dosyasının “kullanıcı kodu” kısmındaki while döngüsünün "yylex fonksiyonu sıfır değeri ile geri dönmediği sürece" devam ettirildiğini görüyorsunuz. yylex atomu bulursa bizim belirlediğimiz değerle geri dönmektedir. Örnek kodda daha sonra yylex’in geri dönüş değerinin switch içerisine sokularak ele alındığını görüyorsunuz.

Bazen kurallarda yapılacak işlemler çok uzun olabilir. Bu durumda bu işlemlerin kurallar kısmında yapılması yerine bir fonksiyona yaptırılması daha uygun olur. Örneğin bir sabitle karşılaşıldığında sabitin istenilen limit dışında olup olmadığı kontrol edilmek istensin. Bu işlem aşağıdaki gibi yapılabilir:

%option noyywrap

%{

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#define CONSTANT 1

#define INVALID\_CONSTANT 2

#define IDENTIFIER 3

int CheckConstant(void);

%}

Constant [\+-]?[0-9]+

Identifier [\_a-zA-Z]+

%%

{Constant} return CheckConstant();

{Identifier} return IDENTIFIER;

\n

.

%%

int CheckConstant(void)

{

double result;

result = atof(yytext);

if (result > LONG\_MAX || result < LONG\_MIN)

return INVALID\_CONSTANT;

return CONSTANT;

}

int main(int argc, char \*argv[])

{

int tokenId;

if (argc > 2 ) {

fprintf(stderr, "wrong number of arguments!\n");

exit(EXIT\_FAILURE);

}

if (argc == 2)

if ((yyin = fopen(argv[1], "r")) == NULL) {

fprintf(stderr, "cannot open file!\n");

exit(EXIT\_FAILURE);

}

while ((tokenId = yylex()) != 0) {

switch (tokenId) {

case CONSTANT:

printf("Constant: %s (%d)\n", yytext, yyleng);

break;

case INVALID\_CONSTANT:

printf("Invalid Constant: %s (%d)\n", yytext, yyleng);

break;

case IDENTIFIER:

printf("Identifier: %s (%d)\n", yytext, yyleng);

break;

}

}

return 0;

}

CheckConstant fonksiyonunun prototip bildiriminin dosyanın "tanımlamalar" kısmında yapıldığına dikkat ediniz. Tabii Flex bildirimleri çok uzunsa biz bu bildirimleri bir başlık dosyasına yerleştirip onu da include edebiliriz. Örneğin:

/\* sample.l \*/

%option noyywrap

%{

#include "sample.h"

%}

Constant [\+-]?[0-9]+

Identifier [\_a-zA-Z]+

%%

{Constant} return CheckConstant();

{Identifier} return IDENTIFIER;

\n

.

%%

int CheckConstant(void)

{

double result;

result = atof(yytext);

if (result > LONG\_MAX || result < LONG\_MIN)

return INVALID\_CONSTANT;

return CONSTANT;

}

int main(int argc, char \*argv[])

{

int tokenId;

if (argc > 2 ) {

fprintf(stderr, "wrong number of arguments!\n");

exit(EXIT\_FAILURE);

}

if (argc == 2)

if ((yyin = fopen(argv[1], "r")) == NULL) {

fprintf(stderr, "cannot open file!\n");

exit(EXIT\_FAILURE);

}

while ((tokenId = yylex()) != 0) {

switch (tokenId) {

case CONSTANT:

printf("Constant: %s (%d)\n", yytext, yyleng);

break;

case INVALID\_CONSTANT:

printf("Invalid Constant: %s (%d)\n", yytext, yyleng);

break;

case IDENTIFIER:

printf("Identifier: %s (%d)\n", yytext, yyleng);

break;

}

}

return 0;

}

/\* sample.h \*/

#ifndef SAMPLE\_H\_

#define SAMPLE\_H\_

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

/\* Symbolic Constants \*/

#define CONSTANT 1

#define INVALID\_CONSTANT 2

#define IDENTIFIER 3

/\* Function Prototypes \*/

int CheckConstant(void);

#endif

Yine derleme işlemi şöyle yapılabilir:

flex –osample.c sample.l

gcc –o sample sample.c (ya da Microsoft derleyicilerinde “cl sample.c”)

Programı çalıştırmadan önce bir test dosyası hazırlamalısınız. Örneğin:

sample test.txt (Linux ve Mac OS X sistemlerinde “./sample test.txt)

**3.5.1.6. Flex’in Ürettiği Kodlarla Programcının Kodlarının Farklı Dosyalarda Bulundurulması**

Flex default olarak içerisinde atomlara ayırma işlemini yapan yylex fonksiyonun bulunduğu “lex.yy.c” isminde bir C kaynak kod dosyası üretmektedir. Anımsanacağı gibi biz Flex dosyasının “Kullanıcı Kodları (User Codes)” bölümünde main fonksiyonunu ve diğer fonksiyonları bulundurabiliyorduk. Aslında bir Flex dosyasının “Kullanıcı Kodları (User Codes)” kısmı tamamen ayrı bir C kaynak dosyasına da taşınabilir. (Bu bölümün Flex dosyasında boş olarak bulundurulabileceğini anımsayınız.) Tabii bizim “kullanıcı kodlarını” ayrı bir dosyaya taşımamız durumunda Flex değişkenlerinin extern bildirimlerinin de bu dosyada bulunması gerekmektedir. Bu bildirimleri tek tek elle yazmak yerine Flex’ten bunların bulunduğu bir başlık dosyasını üretmesini isteyebiliriz. Bu işlem iki biçimde yapılabilmektedir: Birinci seçenek Flex kaynak kodu işleme sokulurken “–header-file=dosya ismi” seçeneğinin eklenmesidir. (Ancak maalesef Flex’in Windows GNU versiyonu henüz bunu desteklememektedir. Bunun için flex’in diğer bir Windows nuyarlaması olan “win\_flex.exe” programını kullanmalısınız) İkinci seçenek Flex kaynak kodunun “Tanımlamalar (Definitions)” kısmında %option header-file=”dosya ismi” tanımlamasının yapılmasıdır. (Ancak maalesef Flex’in Windows GNU versiyonu bunu da henüz desteklememektedir. Bunun için Flex’in diğer Windows sürümü olan “win\_flex.exe” programını kullanmalısınız).

Örneğin bir Flex uygulamasını Visual Studio kullanarak oluşturmak isteyelim (010-FlexWithVisualStudio). Bu işlemi şu adımlardan geçerek yapabiliriz:

1) Flex kaynak dosyası (örneğimizde sampleflex.l) “--header-file=yylex.h” seçeneği ile aşağıdaki gibi işleme sokulur:

win\_flex --wincompat --header-file=lex.yy.h sampleflex.l

Buradan i “lex.yy.c” ve “lex.yy.h” isminde iki dosya elde edilecektir:. “lex.yy.c” dosyasında atomlara ayırma işlemini yapan C kodları, “lex.yy.h” dosyasında ise Flex değişkenlerinin extern bildirimleri bulunacaktır. (Komut satırındaki --wincompat seçeneği ileride ele alınacaktır.)

2) Şimdi “Kullanıcı Kodları (User Codes)” blümündekileri başka bir C dosyasına alabiliriz. (Örneğin bu dosyamızın ismi “samplefleximpl.c” olsun.) Tabii bu dosya içerisinde Flex’in değişkenleri kullanılacağı için bu dosyadan “lex.yy.h” dosyasının include edilmesi gerekmektedir.

3) Eğer programcı isterse yine kendi dosyası için de bir başlık dosyası hazırlayabilir. Mademki Flex’in kendisi de programcının birtakım bildirimlerini kullanmaktadır. O halde bu bildirimler ortak bir başlık dosyasında toplanabilir. (Örneğimizde bu başlık dosyası “samplefleximpl.h” ismindedir.

4) Visual Studio’da yaratılacak projenin içerisine “lex.yy.c” ve “samplefleximpl.c” dosyalarının eklenmesi gerekir. Başlık dosyaları include edildiği için onların doğrudan projeye eklenmesine gerek yoktur. Ancak başlık dosyalarında bir değişiklik yapıldığında kaynak kodların yeniden derlenmesi isteniyorsa bunlar da projeye eklenebilir.

Maalesef Flex’in “win\_flex.exe” Windows uyarlamasında üretilen başlık dosyasında (örneğimizdeki “lex.yy.h” dosyası) UNIX/Linux sistemlerindeki bazı POSIX fonksiyonlarının bulunduğu “unistd.h” dosyası include edilmiştir. Windows sistemlerinde bu dosya olmadığı için derlemede sorunlar oluşabilmektedir. Bunu sorun üretilen “lex.yy.c” ve “samplefleximpl.c” dosyalarının başına YY\_NO\_UNISTD\_H makrosunun define edilmesi ile engellenebilir. (Bu define işlemini Visual Studio’da proje özelliklerindeki “C-C++/Preprocessor” seçeneğiseçeneği ile de yapabilirsiniz.)

Örneğimizde kullanılan dosyalar şöyledir:

/\* sampleflex.l \*/

%option noyywrap

%{

#include "samplefleximpl.h"

%}

Constant [\+-]?[0-9]+

Identifier [\_a-zA-Z]+

%%

{Constant} return CheckConstant();

{Identifier} return IDENTIFIER;

\n

.

%%

/\* samplefleximpl.h \*/

#ifndef SAMPLEFLEXIMPL\_H\_

#define SAMPLEFLEXIMPL\_H\_

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

/\* Symbolic Constants \*/

#define CONSTANT 1

#define INVALID\_CONSTANT 2

#define IDENTIFIER 3

/\* Function Prototypes \*/

int CheckConstant(void);

#endif

/\* samplefleximpl.c \*/

#include <stdio.h>

#include <stdlib.h>

#include "samplefleximpl.h"

#include "lex.yy.h"

int CheckConstant(void)

{

double result;

result = atof(yytext);

if (result > LONG\_MAX || result < LONG\_MIN)

return INVALID\_CONSTANT;

return CONSTANT;

}

int main(int argc, char \*argv[])

{

int tokenId;

if (argc > 2) {

fprintf(stderr, "wrong number of arguments!\n");

exit(EXIT\_FAILURE);

}

if (argc == 2)

if ((yyin = fopen(argv[1], "r")) == NULL) {

fprintf(stderr, "cannot open file!\n");

exit(EXIT\_FAILURE);

}

while ((tokenId = yylex()) != 0) {

switch (tokenId) {

case CONSTANT:

printf("Constant: %s (%d)\n", yytext, yyleng);

break;

case INVALID\_CONSTANT:

printf("Invalid Constant: %s (%d)\n", yytext, yyleng);

break;

case IDENTIFIER:

printf("Identifier: %s (%d)\n", yytext, yyleng);

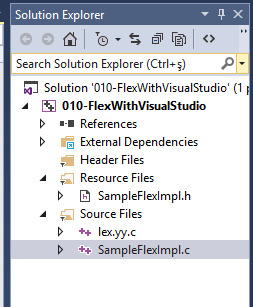
break;

}

}

return 0;

}



Yukarıdaki işlemleri Linux ya da Mac OS X’te de şöyle yapabiliriz. (Aynı dosya isimlerinin küçük harflerle oluşturulduğunu varsayalım)

1) Flex kaynak dosyası aşağıdaki gibi derlenir:

flex --header-file=lex.yy.h sampleflex.l

2) ”samplefleximpl.c” dosyası oluşturulur. Yine ortak bildirimler “samplefleximpl.h” dosyasında bulundurulur. “samplefleximpl.c” dosyasından sampleflex.h ve yylex.h dosyaları include edilir.

3) Oluşturulan bu iki C dosyası aşağıdaki gibi derlenerek birlikte bağlanır:

gcc -o sampleflex yylex.c samplefleximpl.c

**Anahtar Notlar:** Flex ve Bison ile Visual Studio’da hiç komut satırına geçmeden de çalışabilir. Bunun için öncelikle “Flex” ve “Bison” araçlarının Visual Studio build sistemine tanıtılması gerekir. Bu tanıtım Flex ve Bison’un diğer Windows uyarlaması olan “WinFlexBison” dağıtımında önceden hazırlanmış dosyalarla yapılabilmektedir. Visual Studio’da proje seçenekleri üzerinde bağlam menüsünden “Build Dependencies/Build Custimization” seçilir. Buradan da “Find Existing” düğmesi ile “WinFlexBison” dağıtımındaki “custom\_build\_tools” dizinindeki dosyalar seçilir. Artık biz projeye doğrudan “.l” ve “.y” uzantılı dosyaları ekleyebiliriz. Bu dosyalar “win\_flex.exe” ve “win\_bison.exe” programları tarafından çalıştırılacaktır. Tabii burada “win\_flex .exe” ve “win\_bison.exe” programları tarafından üretilen dosyaların da ayrıca projeye manuel olarak eklenmesi gerekir.

**3.5.1.7. C Programlama Dilini Atomlarına Ayıran Flex Kodu**

Internet’te standart pek çok programlama dili için başkaları tarafından yazılmış olan “flex” kodlarını bulabilirsiniz. (Bunun için Google’da “C grammar lex” gibi bir arama yapılabilir.) Örneğin <https://www.lysator.liu.se/c/ANSI-C-grammar-l.html#comment> sitesinde “ ANSI C Grammar, Lex Specification” isimli başlık altında aşağıdaki gibi bir flex (ya da lex) kodu verilmiştir.

D [0-9]

L [a-zA-Z\_]

H [a-fA-F0-9]

E [Ee][+-]?{D}+

FS (f|F|l|L)

IS (u|U|l|L)\*

%{

#include <stdio.h>

#include "y.tab.h"

void [count()](https://www.lysator.liu.se/c/ANSI-C-grammar-l.html#count);

%}

%%

"/\*" { [comment()](https://www.lysator.liu.se/c/ANSI-C-grammar-l.html#comment); }

"auto" { count(); return(AUTO); }

"break" { count(); return(BREAK); }

"case" { count(); return(CASE); }

"char" { count(); return(CHAR); }

"const" { count(); return(CONST); }

"continue" { count(); return(CONTINUE); }

"default" { count(); return(DEFAULT); }

"do" { count(); return(DO); }

"double" { count(); return(DOUBLE); }

"else" { count(); return(ELSE); }

"enum" { count(); return(ENUM); }

"extern" { count(); return(EXTERN); }

"float" { count(); return(FLOAT); }

"for" { count(); return(FOR); }

"goto" { count(); return(GOTO); }

"if" { count(); return(IF); }

"int" { count(); return(INT); }

"long" { count(); return(LONG); }

"register" { count(); return(REGISTER); }

"return" { count(); return(RETURN); }

"short" { count(); return(SHORT); }

"signed" { count(); return(SIGNED); }

"sizeof" { count(); return(SIZEOF); }

"static" { count(); return(STATIC); }

"struct" { count(); return(STRUCT); }

"switch" { count(); return(SWITCH); }

"typedef" { count(); return(TYPEDEF); }

"union" { count(); return(UNION); }

"unsigned" { count(); return(UNSIGNED); }

"void" { count(); return(VOID); }

"volatile" { count(); return(VOLATILE); }

"while" { count(); return(WHILE); }

{L}({L}|{D})\* { count(); return([check\_type()](https://www.lysator.liu.se/c/ANSI-C-grammar-l.html#check-type)); }

0[xX]{H}+{IS}? { count(); return(CONSTANT); }

0{D}+{IS}? { count(); return(CONSTANT); }

{D}+{IS}? { count(); return(CONSTANT); }

L?'(\\.|[^\\'])+' { count(); return(CONSTANT); }

{D}+{E}{FS}? { count(); return(CONSTANT); }

{D}\*"."{D}+({E})?{FS}? { count(); return(CONSTANT); }

{D}+"."{D}\*({E})?{FS}? { count(); return(CONSTANT); }

L?\"(\\.|[^\\"])\*\" { count(); return(STRING\_LITERAL); }

"..." { count(); return(ELLIPSIS); }

">>=" { count(); return(RIGHT\_ASSIGN); }

"<<=" { count(); return(LEFT\_ASSIGN); }

"+=" { count(); return(ADD\_ASSIGN); }

"-=" { count(); return(SUB\_ASSIGN); }

"\*=" { count(); return(MUL\_ASSIGN); }

"/=" { count(); return(DIV\_ASSIGN); }

"%=" { count(); return(MOD\_ASSIGN); }

"&=" { count(); return(AND\_ASSIGN); }

"^=" { count(); return(XOR\_ASSIGN); }

"|=" { count(); return(OR\_ASSIGN); }

">>" { count(); return(RIGHT\_OP); }

"<<" { count(); return(LEFT\_OP); }

"++" { count(); return(INC\_OP); }

"--" { count(); return(DEC\_OP); }

"->" { count(); return(PTR\_OP); }

"&&" { count(); return(AND\_OP); }

"||" { count(); return(OR\_OP); }

"<=" { count(); return(LE\_OP); }

">=" { count(); return(GE\_OP); }

"==" { count(); return(EQ\_OP); }

"!=" { count(); return(NE\_OP); }

";" { count(); return(';'); }

("{"|"<%") { count(); return('{'); }

("}"|"%>") { count(); return('}'); }

"," { count(); return(','); }

":" { count(); return(':'); }

"=" { count(); return('='); }

"(" { count(); return('('); }

")" { count(); return(')'); }

("["|"<:") { count(); return('['); }

("]"|":>") { count(); return(']'); }

"." { count(); return('.'); }

"&" { count(); return('&'); }

"!" { count(); return('!'); }

"~" { count(); return('~'); }

"-" { count(); return('-'); }

"+" { count(); return('+'); }

"\*" { count(); return('\*'); }

"/" { count(); return('/'); }

"%" { count(); return('%'); }

"<" { count(); return('<'); }

">" { count(); return('>'); }

"^" { count(); return('^'); }

"|" { count(); return('|'); }

"?" { count(); return('?'); }

[ \t\v\n\f] { count(); }

. { /\* ignore bad characters \*/ }

%%

yywrap()

{

return(1);

}

comment()

{

char c, c1;

loop:

while ((c = input()) != '\*' && c != 0)

putchar(c);

if ((c1 = input()) != '/' && c != 0)

{

unput(c1);

goto loop;

}

if (c != 0)

putchar(c1);

}

int column = 0;

void count()

{

int i;

for (i = 0; yytext[i] != '\0'; i++)

if (yytext[i] == '\n')

column = 0;

else if (yytext[i] == '\t')

column += 8 - (column % 8);

else

column++;

ECHO;

}

int check\_type()

{

/\*

\* pseudo code --- this is what it should check

\*

\* if (yytext == type\_name)

\* return(TYPE\_NAME);

\*

\* return(IDENTIFIER);

\*/

/\*

\* it actually will only return IDENTIFIER

\*/

return(IDENTIFIER);

}

Yukarıdaki Flex programını kullanan bir main fonksiyonu şöyle yazılabilir (“Src/011-CProgrammingLanguageFlex”):

#include <stdio.h>

#include <stdlib.h>

#include "cflex.h"

#include "lex.yy.h"

int main(int argc, char \*argv[])

{

int tokenId;

if (argc > 2) {

fprintf(stderr, "wrong number of arguments!\n");

exit(EXIT\_FAILURE);

}

if (argc == 2)

if ((yyin = fopen(argv[1], "r")) == NULL) {

fprintf(stderr, "cannot open file!\n");

exit(EXIT\_FAILURE);

}

while ((tokenId = yylex()) != 0)

printf("%s\n", yytext);

return 0;

}

İlgili Flex dosyasında tek karakterli atomların id’lerinin doğrudan o karakterlerin karakter tablosundaki sayısal değeri olarak alındığına dikkat ediniz. Bu durum kodlama bakımından bazı pratiklikler sağlamaktadır.

**3.5.1.8. Flex’in Bazı Ayrıntıları**

Flex normal olarak yyin ile belirtilen dosyadaki karakterleri atomlarına ayırmaya çalışır. (Anımsanacağı gibi default durumda yyin değişkeni “stdin” dosyasını belirtiyordu. İşte Flex bize yyrestart isimli bir fonksiyon da vermektedir. Bu fonksiyonun prototipi şöyledir:

void yyrestart(FILE \*f);

Fonksiyon parametre olarak FILE \* türünden dosya bilgi göstericisini alır. yyrestart fonksiyonu çağrıldığında artık yylex bu yeni dosyanın başından itibaren işlemine devam eder. Böylece bir Flex dosyası bittiğinde işlemlerin başka bir Flex dosyasından devam ettirilmesi sağlanabilmektedir:

%option noyywrap

%{

#include <stdio.h>

#include <stdlib.h>

int g\_lines;

int g\_words;

int g\_chars;

%}

%%

[0-9A-Za-z]+ { ++g\_words; g\_chars += yyleng;}

. { ++g\_chars; }

\n { ++g\_lines; ++g\_chars; }

%%

int main(char argc, char \*argv[])

{

int i;

FILE \*f;

if (argc == 1) {

yylex();

printf("%d %d %d\n", g\_lines, g\_words, g\_chars);

exit(EXIT\_SUCCESS);

}

for (i = 1; i < argc; ++i) {

if ((f = fopen(argv[i], "r")) == NULL) {

fprintf(stderr, "cannot read file!..\n");

exit(EXIT\_FAILURE);

}

yyrestart(f);

yylex();

printf("%d %d %d %s\n", g\_lines, g\_words, g\_chars, argv[i]);

g\_words = g\_lines = g\_chars = 0;

}

return 0;

}

Flex’in yywrap isimli fonksiyonundan daha önce söz etmiştik. Bu fonksiyon dosyanın sonuna gelindiğinde yylex tarafından çağrılmaktadır. Bazen dosya sonuna gelindiğinde bazı ayarlamaların yapılması istenebilir. İşte yywrap fonksiyonu bu amaçla kullanılmaktadır. Bu fonksiyonu yazan programcı eğer 0 ile geri dönerse yylex işlemine devam eder. (Örneğin bunun içerisinde yyrestart yapılmış olabilir.) Eğer sıfır dışı bir değerle geri dönerse yylex işlemini bitirir. Ancak biz yylex fonksiyonunun yywrap fonksiyonunu çağırmasını da Flex kaynak dosyasının başına “%option noyywrap” direktifini yerleştirerek engelleyebiliriz. Bu durumda yylex yywrap fonksiyonunu çağırmayacağı için bizim de bu fonksiyonu tanımlamamıza gerek kalmaz.

Flex’in ürettiği C kodu normal olarak atomlara ayıracağı dosyayı tamponlamaktadır. Yani Flex’in ürettiği kod dosyayı karakter karakter okumak yerine onun bir bölümünü belleğe çekip karakterleri oradan hızlı bir biçimde almaktadır. İşte biz bazen onun kullandığı bu tamponun değiştirilmesini de isteyebiliriz. Tampon YY\_BUFFER\_STATE isimli bir türle tespit edilmiştir. Belli bir uzunlukta yeni bir tampon yaratmak için yy\_create\_buffer fonksiyonu kullanılmaktadır. Örneğin:

YY\_BUFFER\_STATE bp;

...

bp = yy\_create\_buffer(yyin, 4096 );

Burada yy\_create\_buffer fonksiyonunun iki parametre aldığına dikkat ediniz. Birinci parametre FILE \* türündendir ve atomlara ayrılacak dosyaya ilişkin dosya bilgi göstericisini alır, ikinci parametre ise tamponun uzunluğunu belirtmektedir. yylex’in yaratılan bu yeni tamponu kullanması da yy\_switch\_tobuffer fonksiyonuyla sağlanmaktadır. Örneğin:

yy\_switch\_to\_buffer(bp);

Flex’in ürettiği yylex fonksiyonunun atomlarına ayıracağı yazı her zaman bir dosyada mı bulunmak zorundadır? Yanıt hayır. İşte yy\_scan\_string isimli fonksiyon yylex’in dosyadan değil de sonu ‘\0’ ile biten yazıyı atomlarına ayırmasını sağlamaktadır. Fonksiyonun prototipi şöyledir:

YY\_BUFFER\_STATE yy\_scan\_string(yyconst char \*yy\_str);

yy\_scan\_string fonksiyonunun kullanımına ilişkin şöyle bir örnek verebiliriz:

%option noyywrap

%{

#include <stdio.h>

#include <stdlib.h>

int g\_lines;

int g\_words;

int g\_chars;

%}

%%

[0-9A-Za-z]+ { ++g\_words; g\_chars += yyleng;}

. { ++g\_chars; }

\n { ++g\_lines; ++g\_chars; }

%%

int main(void)

{

char str[] = "bu bir denemedir\nevet denemedir";

YY\_BUFFER\_STATE bs;

bs = yy\_scan\_string(str);

yy\_switch\_to\_buffer(bs);

yylex();

printf("%d %d %d\n", g\_lines, g\_words, g\_chars);

return 0;

}

yy\_scan\_string fonksiyonun yanı sıra sonu ‘\0’ karakter ile bitmeyen yazıların atomların a ayrılması için yy\_scan\_buffer isimli bir fonksiyon da bulundurulmaktadır. Bu fonksiyon belli bir adresten belli miktarda karakterin taranmasını sağlar:

YY\_BUFFER\_STATE yy\_scan\_buffer(char \*base, yy\_size\_t size);

Flex ile ilgili daha başka ayrıntılar da vardır. Ancak biz burada bu ayrıntıları ele almayacağız. Bunlar için Flex’in dokümanlarına bakabilirsiniz.

**3.6. Derleyiciler ve Yorumlayıcılarda String Tablolarının Oluşturulması**

Pek çok derleyici ya da yorumlayıcı ayrıştırdığı atomlar içerisinde değişkenleri ve sabitleri string tablosu denilen bir tabloya yerleştirmektedir. Örneğin aşağıdaki gibi bir C kodu atomlarına ayrılacak olsun:

int count, total;

count = 10;

total = 0;

Bu kod içerisinde count ve total değişken (identifer) isimlerinin birden fazla kez kullanıldığını görüyorsunuz. İşte lexical analiz modülü tarafından bu isimler ilk görüldüğünde string tablosu denilen bir tabloya yerleştirilmekte ve yeniden görüldüklerinde de artık o tablodan elde edilmektedir. Tabii aslında string tablolarında değişken isimleri doğrudan tutulmamaktadır. Değişken isimleri dinamik olarak tahsis edilen bir alana kopyalanmakta, string tablolarında yalnızca onların adresleri tutulmaktadır. String tablolarına yalnızca değişken isimleri değil sabitler de yerleştirilebilmektedir.

String tablolarının kullanılma gerekçesi iki maddeyle özetlenebilir:

1) String tabloları sayesinde aynı değişken isimlerinin ve sabitlerin gereksiz biçimde birden fazla kez parse ağacı içerisinde (ya da sembol tabloları) yer kaplaması engellenmiş olur.

2) String tablolarında her değişken ismi yalnızca bşir kez bulunduğundan parser modülü değişkenleri onları isimsel olarak yalnızca adreslerini kullanrak karşılaştırabilmektedir.

String tablolarının gerçekleştirilmesinde veri yapısı olarak genellikle “hash tabloları (hash tables)” kullanılmaktadır. (Çünkü hash tabloları arama konusunda oldukça hızlıdır. “Hash Tabloları” konusu “Sistem Programlama ve İleri C Uygulamaları-1” isimli kursta ele alınmıştı.) Burada bir uyarıda bulıunmak istiyoruz: String tablosunu “Sembol Tablosu (Symbol Table)” ile karıştırmamalısınız. Sembol tabloları “parser” modülü tarafından kullanılmaktadır ve sembol tablolarında değişkenlerin türleri ve diğer özellikleri tutulmaktadır.

Örnek bir string tablosu şöyle gerçekleştirilebilir:

/\* StringTable.h \*/

#ifndef STRINGTABLE\_H\_

#define STRINGTABLE\_H\_

#include <stddef.h>

/\* Symbolic Constants \*/

#define TABLE\_SIZE 1000

/\* Type Declarations \*/

typedef struct tagSTRNODE {

char \*str;

size\_t size;

struct tagSTRNODE \*next;

} STRNODE;

/\* Function Prototypes \*/

void InitStringTable(void);

char \*LookupStrSize(const char \*str, size\_t size);

char \*LookupStr(const char \*str);

void DestroyStringTable(void);

#endif

/\* StringTable.c \*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "StringTable.h"

/\* static Function prtotypes \*/

static size\_t hashFunc(const char \*str, size\_t size);

/\* Global Data Definitions \*/

STRNODE \*g\_hashTable[TABLE\_SIZE];

/\* Function Definitions \*/

static size\_t hashFunc(const char \*str, size\_t size)

{

unsigned long hash = 5381;

int i;

for (i = 0; i < size; ++i)

hash = ((hash << 5) + hash) + str[i];

return hash % TABLE\_SIZE;

}

void InitStringTable(void)

{

int i;

for (i = 0; i < TABLE\_SIZE; ++i)

g\_hashTable[i] = NULL;

}

char \*LookupStrSize(const char \*str, size\_t size)

{

size\_t hash;

STRNODE \*node;

hash = hashFunc(str, size);

node = g\_hashTable[hash];

while (node != NULL) {

if (size == node->size && !strncmp(str, node->str, size))

return printf("bulundu\n"), node->str;

node = node->next;

}

if ((node = (STRNODE \*)malloc(sizeof(STRNODE))) == NULL)

return NULL;

if ((node->str = (char \*)malloc(size + 1)) == NULL) {

free(node);

return NULL;

}

strncpy(node->str, str, size + 1);

node->size = size;

node->next = g\_hashTable[hash];

g\_hashTable[hash] = node;

return node->str;

}

char \*LookupStr(const char \*str)

{

char \*end = str;

while (\*end != '\0')

++end;

return LookupStrSize(str, (size\_t)(end - str));

}

void DestroyStringTable(void)

{

int i;

STRNODE \*node, \*tempNode;

for (i = 0; i < TABLE\_SIZE; ++i) {

node = g\_hashTable[i];

while (node != NULL) {

tempNode = node->next;

free(node->str);

free(node);

node = tempNode;

}

}

}

#if 1

int main(void)

{

char text[1024];

char \*str;

InitStringTable();

for (;;) {

printf("Bir yazi giriniz:");

gets(text);

if (!strcmp(text, "quit"))

break;

if ((str = LookupStr(text)) == NULL) {

fprintf(stderr, "cannot lookup string!..\n");

exit(EXIT\_FAILURE);

}

printf("%s (%x)\n", str, str);

}

DestroyStringTable();

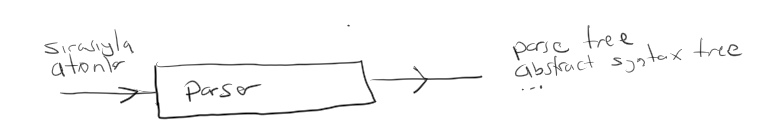
return 0;

}

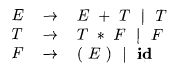
#endif

**3.7. Parse İşlemleri (Parsing)**

“Parsing” İngilizce “ayrıştırma”, “gramer olarak çözümleme” anlamına gelen bir sözcüktür. Bilgisayar bilimlerinde bir yazının (biçimsel dil terminolojisinde bir string’in) verilen bir gramere (sentaksa) uygunluğunun denetlenmesi ve bunun bir veri yapısı biçiminde ifade edilmesi sürecine “parsing” denilmektedir. Bu işi yapan dil işlemcileri (language processors) de “parser” olarak isimlendirilir. Parser modülünün giridisi atomlardır. (Örneğin bu girdi “lexical analiz” aşamasında gerçekleştirilen GetNextToken gibi bir fonksiyonla ya da flex’in oluşturduğu yylex fonksiyonuyla elde edilebilir.) Parser modülünün çıktısı ise ilgili girdinin gramere göre çözümlenerek bir veri yapısı haline getirilmiş biçimidir. Bu çıktı da genellikle “parse ağacı (parse tree)” ya da “soyut sentaks ağacı (abstract syntax tree)” biçiminde isimlendirilir. Tabii derleyicilerin ve yorumlayıcıların parser modülleri grameri temsil eden başka bir veri yapısı da üretebilir.



Şüphesiz “parse” işlemi için öncelikle elimizde biçimsel (resmi) bir gramerin (formal grammar) üretici bir gramer (generative grammar) biçiminde ifade ifade edilmesi gerekir. Dilin üretici grameri (sentaksı) matematiksel bir biçimde ya da BNF notasyonuyla betimlenebilmektedir. Biz burada üretici grameri bazen matematiksel gösterimle bazen de BNF notasyonuyla ifade edeceğiz. Örneğin aşağıda bir üretici gramerin matematiksel olarak gösterimini görüyorsunuz (örnek “Compiler Principles and Techniques” isimli kitabından alınmıştır):



Burada son semboller (terminal symbols) ya da atomlar “id”, “+”, “\*”, “(“ ve “)” karakterlerinden oluşmaktadır. E, T ve F ise ara sembollerdir. Matematiksel gösterimde (ve BNF notasyonunda) ‘|’ karakterinin “veya” anlamına gelen bir meta karakter olduğunu anımsayınız. Aynı üretici gramer BNF notasyonu ile aşağıdaki gibi ifade edilebilir.

E: E ‘+’ T

T

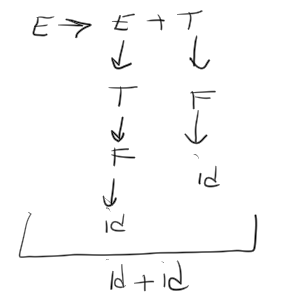
T: T ‘\*’ F

F

F: ‘(‘ E ‘)’

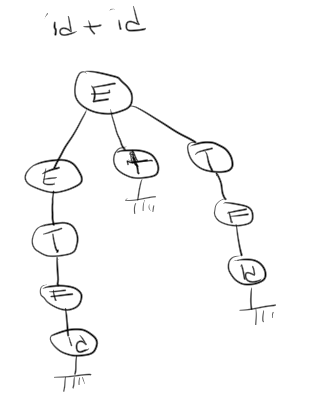
‘id’

Örneğin bu gramere göre üretilen bir çıktı (string) şöyle olabilir:

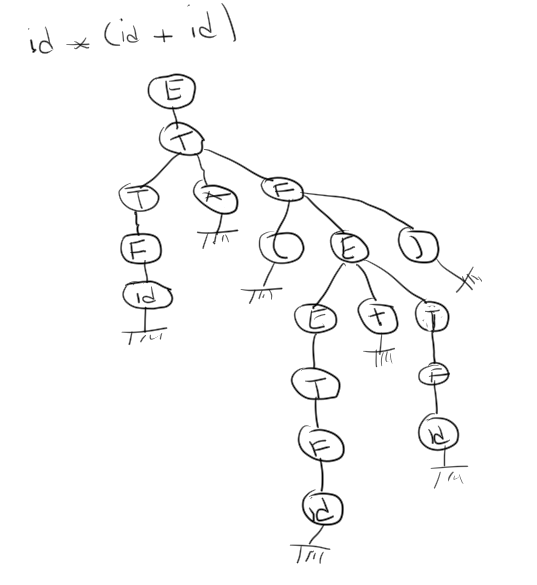


Bu gramerin örneğin id \* (id + id) gibi bir string de üretebileceğine dikkat ediniz.

Parse ağacı (“Somut Sentaks Ağacı (“Concrete Sytax Tree”) de denilmektedir) bir string’in üretici gramere göre açılımını gösteren bir veri yapısıdır. Parse ağacında ara semboller ve son semboller (atomların) birer düğüm olarak ifade edilirler. Parse ağacının yapraklarında (leaves) son semboller (atomlar) bulunur. Örneğin yukarıdaki gramerde “id + id” yazısı için oluşturulabilecek parse ağacı şöyledir:

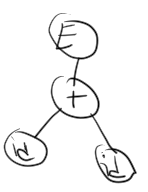


Şimdi de “id \* (id + id)” string’i için parse ağacını oluşturalım:

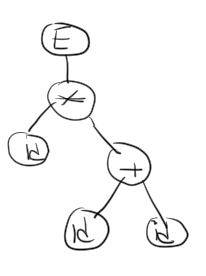


Bir string için tek bir “parse ağacı” oluşturulabileceğine dikkat ediniz.

Soyut sentaks ağacı (abstract syntax tree) gereksiz kural geçişlerinin elimine edildiği, parse ağacının daha sade hale getirilmiş bir biçimidir. Soyut sentaks ağacında son semboller yapraklarda bulunmak zorunda değildir. Bundan dolayı soyut sentaks ağacı “parser” modülünün çıktısı olmaya daha adaydır. Ancak soyut sentaks ağacı kuralları belirlenmiş olan, her zaman aynı biçimde oluşturulan resmi bir ağaç değildir. Dolayısıyla aynı string için farklı programcılar farklı soyut sentaks ağaçları oluşturabilirler. Örneğin “id + id” string’i için aşağıdaki gibi bir soyut sentaks ağacı oluşturulabilir:

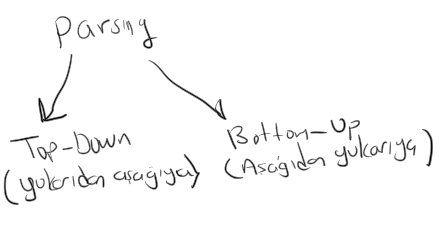


Görüldüğü gibi soyut sentaks ağacında gereksiz geçişler elimine edilmiştir. Şimdi de “id \* (id + id)” string’i için soyut sentaks ağacı oluşturalım:



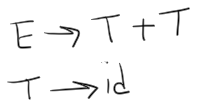
**3.7.1. Parse İşlemleri İçin Teorik Temel**

Parse işlemleri atomların sırasıyla elde edilip incelenmesi yoluyla yapılmaktadır. Tabii bu süreç için pek çok teknik oluşturulmuştur. Genel olarak parse işlemi “yukarıdan aşağıya (top down)” ya da “aşağıdan yukarayıa (bottom up)” denilen iki temel teknikle yapılabilmektedir.

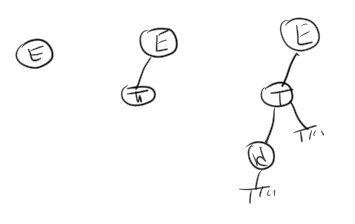


**3.7.1.1. Yukarıdan Aşağıya Parse (Top-Down Parse) İşlemi**

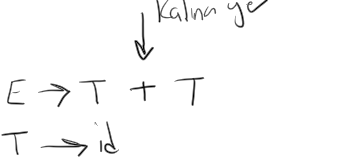
Yukarıdan aşağıya parse işleminde string’te sıradaki atom alınır ve bu atom bulunana kadar üretici gramerde aşağıya doğru inilir. Sonra diğer atom alınır bu atom da bulunana kadar kalının yerden aşağıda doğru inilir. Örneğin aşağıdaki gibi basit bir üretici gramer söz konusu olsun:



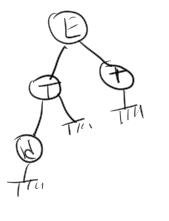
Şimdi biz “id + id” biçiminde bir string’i yukarıdan aşağıya doğru parse etmek isteyelim. Burada önce ilk atom olan id’nin yerini bulmak gerekir. Bunun için gramerin tepesinden sırasıyla id bulunana kadar aşağıya inilir (zaten “yukarıdan aşağıya (top-down) parse” ismi buradan gelmektedir):



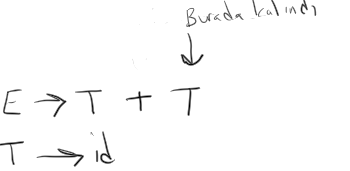
Pekiyi ilk “id” bulunduktan sonra ağaçta kalınan yer neresidir? T’den başka gidilecek yer olmadığına göre kalınan yer E’de T’den sonraki yerdir.

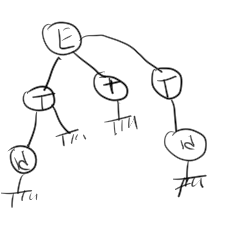


Şimdi ikinci atom çekilir. Bu atom ‘+’ atomudur. Bu atom da kalınan yerden itibaren aşağıya doğru bulunmaya çalışılır ki, zaten birinci kuralda bulunacaktır:

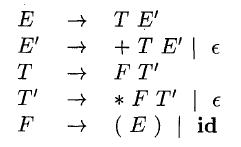


Şimdi sıradaki atom yine elde edilir. Kalınan yer E’deki ‘+’dan sonraki yerdir:

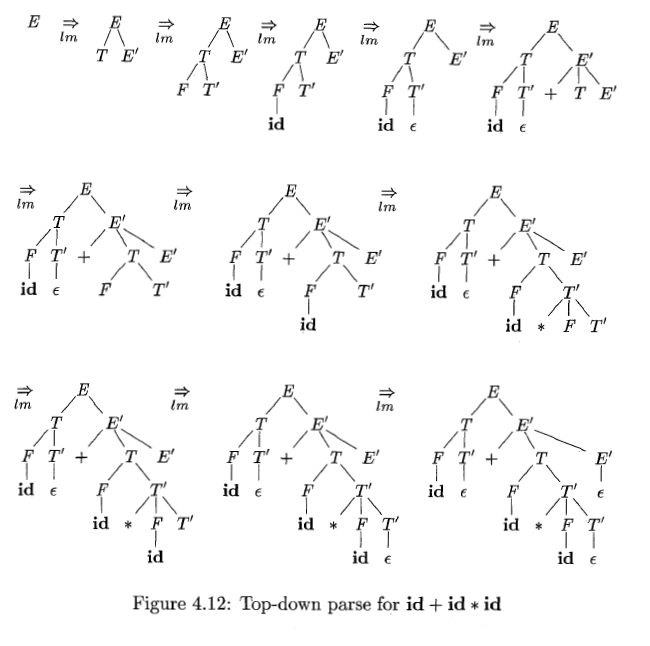




Yukarıdan aşağıya parse işlemini şöyle de anlatabiliriz: Atomlar sırasıyla elde edilir, üretici gramerin tepesinden aşağıya inilerek bunların yeri bulunur ve ağaç oluşturulur. Sonraki atomlar hep kalınan yerden itibaren aranmaktadır. Bu yöntemde parse ağacının oluşturulması “depth first” dolaşım sistemine uygundur. (Buna İngilizce “leftmost derivation” da denilmektedir.) Aşağıda Aho’nun “Compiler Design and Implementation” kitabından alınmış bir “yukarıdan aşağıya parse” örneğini görüyorsunuz:



Burada epsilon boş küme anlamına gelmektedir. Yani epsilon görülğünde artık o ara sembol sonlandırılmış olur. Şimdi bu üretici gramer için “id + id \* id” string’inin parse ağacını oluşturalım:

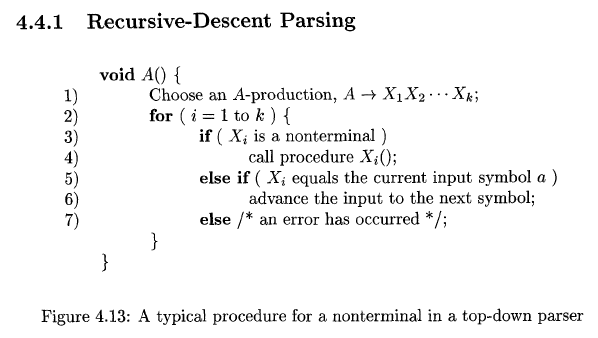


Pekiyi böyle bir yukarıdan aşağıya parse işlemini yapan genel bir kod nasıl yazılabilir? Aslında yukarıdan aşağıya parse işlemi basit bir biçimde şöyle genelleştirilebilir:

- Tüm ara semboller (non-terminal symbols) için birer fonksiyon yazılır. Örneğin yukarıdaki gramerde T, E’, F, T’ için birer fonksiyon yazılır. (İstenirse son semboller için de fonksiyonlar yazılabilir.)

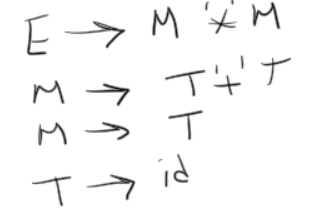
- Ara sembollere ilişkin kurallardaki ara sembollere ilişkin fonksiyonlar bir diziye yerleştirilir.

- Çözümlenecek string’ten sıradaki atom elde edilir. Bu atom kök ara sembolden başlayarak fonksiyon çağrılarıyla bulunmaya çalışılır. Atom bulunduğunda string’teki sonraki atomdan devam edilir. Zaten fonksiyon çağrıları kalınan yerden devamı kendiliğinden sağlayacaktır. Aho’nun “Compiler Design and Implementation” kitabında bu işlemin semboli kodları (pseudo codes) şöyle verilmiştir:



Bu algoritmayla oluşturulan parser’a “recursive-descent parser” da denilmektedir. Bu algoritmanın basit bir gramere uygulanmış hali kurs notlarında “Src\015-RecursiveDescentParser” klasöründe bulunmaktadır.

Pekiyi bir kuralın sol tarafında aynı ara sembolden birden fazla kez bulunduğu durumda (yani seçeneklerin söz konusu olduğu durumda) hangi ara sembole gitmek gerekir? Örneğin:



Burada M ara sembolü iki seçeneklidir. Yukarıdan aşağıya parse işleminde önce bunlardan biri denenip gramerle uyuşum olmadığında diğerini denemek akla uygun gelebilir. Bu stratejiye “parsing” terminolojisinde “geriye dönme (backtracking)” denilmektedir. Şüphesiz geriye dönme performansı azaltıcı bir unsur oluşturur. Seçenekli durumlarda geriye dönme sonraki birkaç atoma bakılarak elimine edilebilmekteedir. Böylece daha işin başında doğru seçeneğe yönelim sağlanmaktadır. İşte bu tür parser’lara “ön kestirimli (predicative) yukarıdan aşağıya parser”lar denilmektedir. Ön kestirimli yukarıdan aşağıya parser’lar sıklıkla “LL(k)” biçiminde gösterilirler. Burada k kaç ileriye bakılacağını belirtmektedir. Bağlam bağımsız gramerlerin çoğu yukarıdan aşağıya yalnızca bir sonraki atoma bakılarak yani LL(1) parser’larla parse edilebilmektedir. (Özel olarak k tane sonraki atoma bakılarak parse edilebilen gramerlere de LL(k) gramer denilmektedir. LL(k) teriminde ilk L “Left To Right” ikinci L ise “Left Most Derivation” sözcüklerinden gelmektedir.)

**3.7.1.2. Aşağıdan Yukarıya Parse (Bottom-Up Parse) İşlemi**

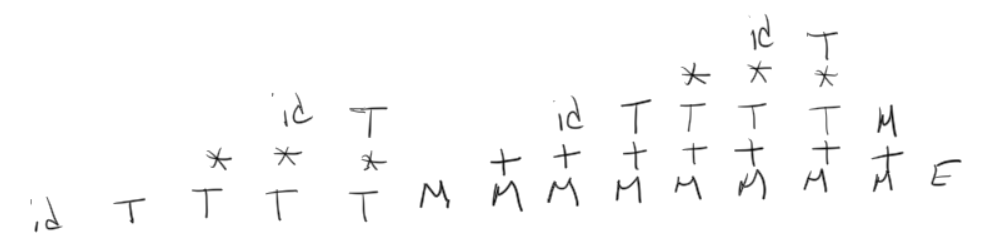
Aşağıdan yukarıya parse işleminde tam ters bir yol izlenmektedir. Sıradaki atom alınır, gramerin en aşağısından başlayarak üste doğru çıkılıp gramere uygunluk denetlenir. Bu denetleme sırasında bir stack’e gereksinim duyulduğu için bu yönteme “Shift-Reduce” Parsing de denilmektedir. (Buradaki Shift “push” olarak, Ruduce ise “pop” olarak düşünülebilir. Tipik olarak aşağıdan yukarıya parse işlemi şu sistematikle yapılmaktadır: Sıradaki atom alınır, aşağıdan başlanarak gramere uydurulmaya çalışılır. Uymazsa stack’a push (shift) edilir. Sonra yeni atom alınır. Bu atomda da aynı işlemler yapılır. Ancak stack’e push işleminden sonra stack’teki durum eğer bir indirgeme (reduce) gerektiriyorsa (yani bir ara sembolle ifade edilebiliyorsa) stack’ten indirgenecek semboller alınır ve indirgenmiş yeni ara sembol stack’e atılır. Bu biçimde ilerlenerek kök sembole ulaşılmaya çalışılır. Örneğin aşağıdaki gibi bir gramer olsun:

E -> M ‘+’ M

M -> T ‘\*’ T

T -> id

Bu gramer için “id \* id + id \* id” string’i verilmiş olsun, “shift” ve “reduce” işlemleri aşağıdaki gibi yapılmaktadır:



Aşağıdan yukarıya parse işlemi de bir kuralda alternatifler olduğunda “geri dönmeyi (backtracking)” gerektirebilmektedir. Geriye dönme işlemini ortadan kaldırabilmek için yine burada da sonraki atomun ele alınması gerekebilir. Aşağıdan yukarıyya parse işleminin atomların soldan sağa alınarak yapılmasından dolayı bu tür parser’lara “LR Parser” da denilmektedir. (Buradaki L harfi “Left To Right” sözcüğünden, R harfi ise “Rightmost derivation” sözcüğünden gelmektedir. LR parser’lar alternatif durumlarında ilerideki kaç elemana bakılacağına göre LR(k) biçiminde isimlendirilmektedir. Pek çok bağlam bağımsız dil LR(1) aşağıdan yukarıya parser’larla ayrıştırılabilmektedir.

LR parser’ların özel bir türüne LALR parser’lar denilmektedir. Örneğin Bison LALR(1) tarzı parser kullanmaktadır. Bu tür parser’lar için teorik bilgi için önerilen kaynaklara başvurabilirsiniz.

**Parse İşlemininin Yapılmasına Ön Ayak Olan Araçlar**

Nasıl flex (eski ismiyle lex) atomlarına ayırma işinde bize yardımcı oluyorsa parse işleminde de çeşitli yardımcı araçlardan faydalanılabilinmektedir. Genel olarak bu araçlara “parser generator” denilmektedir. Bu araçlar bizden dilin gramerini BNF ve türevleriyle alırlar ve parse işlemini yapan kodu üretirler. Bu kod derlenerek amaç doğrultusunda kullanılabilmektedir. Bu araçlardan en eskisi ve ünlüsü “yacc (yet another compiler compiler)” isimli araçtır. GNU projesi kapsamında “yacc” aracının “bison” isimli daha modern bir versiyonu oluşturulmuştur. Bison büyük ölçüde “yacc” ile uyumludur. Bunların dışında yine “ANTLR” isimli araç parse işlemleri için oldukça yoğun kullanılmaktadır. Maalesef ANTLR’nin scanner ve parser üreticileri “Java”, “C#” dillerine ilişkin kodları üretmesine karşın bunların C ve C++ versiyonları henüz etkin biçimde oluşturulamamıştır. Ancak çalışmalar devam etmektedir. Bunların dışında da daha az tercih edilen çeşitli “parser generator” araçları mevcuttur.

**Bison Aracının Kullanımı**

Bison flex ile beraber kullanılabilecek biçimde tasarlanmıştır. Bison parse işlemini yaparken default olarak atomları yylex isimli fonksiyonu çağırarak elde eder. Bison’un kurulumu yine oldukça basittir. <https://www.gnu.org/software/bison/> sitesinden bison indirilebilir. Fakat zeten Linux sistemlerinde Bison tıpkı Flex gibi temel bir araçtır ve default kurulumda zaten yüklü olarak bulunmaktadır. Bison’un Windows versiyonu için yine seçenek vardır. Birincisi GNU’nun kendi yazılımıdır. Bu <http://gnuwin32.sourceforge.net/packages/bison.htm> adresinden indirilebilir. İkincisi de WinFlexBison aracıdır. Bu araçta bison programı “win\_bison.exe” dosyası ile çalıştırılmaktadır.

Bir bison dosyası tıpkı Flex’te olduğu gibi üç bölüme ayrılmaktadır:

%{

Giriş (Prologue)

%}

Bison Bildirimleri (Bison Declarations)

%%

Gramer Kuralları (Grammar Rules)

%%

Kullanıcı Kodları (User Codes / Epilogue)

Bison dosyalarının uzantıları geleneksel olarak “.y” biçimindedir. Tıpkı Flex’te olduğu gibi Bison dosyaları da iki tane %% karakterleri ile üç bölüme ayrılmıştır. Dosyanın “Giriş” bölümünde tıpkı Flex’te olduğu gibi %{ .. %} atomlarının arasına C’ce anlamlı olabilecek bildirimler ve tanımlamalar yerleştirilir. Bu bölüme yerleştirilen kodlar yine Bison’un ürettiği C kaynak dosyasının başına yerleştirilmektedir. Yine ilk %% atomunun yukarısındaki bölüme %{ ... %} bloğunun dışına Bison’ca anlamlı % karakteri ile başlayan bildirimler yerleştirilebilmektedir. Bu bildirimlerin bazıları ilerleyen kısımlara ele alınacaktır. Aslında ilk %% karakterlerinin yukarısındaki “Giriş” ve “Bison Bildirimleri” kısımları birden fazla kez bu bölüm içerisinde bulunabilir.

İki tane %% atomunun arasına gramer kuralları klasik BNF notasyonuna göre yerleştirilir. Bir ara sembolik seçenekleri ‘|’ karakteriyle belirtilmektedir. Kurallar birden fazla satıra yayılmış olarak yazılabilirler. Ancak her satıra bir kuralın yazılması seçenek durumunda da seçeneklerin ayrı bir satıra yazılması iyi bir tekniktir. Kurallarda son semboller (ya da atomlar) tek tırnak ya da çift tırnak ile belirtilebilmektedir. Dosyanın “Kullanıcı Kodları” bölümüne istenildiği kadar C kodu yerleştirilebilir. Örneğin tipik main fonksiyonu burada tanımlanmalıdır. Parse işlemini yapan fonksiyon yyparse isimli fonksiyondur. Dolayısıyla main fonksiyonu içerisinde bu fonksiyonun çağrılması gerekmektedir.

Tipik olarak bir bison programı şöyle işleme sokulmaktadır:

bison sample.y (ya da win\_bison sample.y)

Bison programı dosyanın ismi “x.y” olmak üzere bize default olarak “x.tab.c” dosyasını üretir. Ancak “-o” komut satırı seçeneği ile biz dosyanın ismini de istediğimiz ayarlayabiliriz.

Mademki Bison atomları elde etmek için yylex fonksiyonunu çağırmaktadır. Bu durumda kodun en azından bir yylex fonksiyonuna sahip olması gerekir. Fakat programcı zaten yylex fonksiyonunu çoğu kez Flex programı tarafından oluşturmaktadır. Bu durumda Bison tarafından üretilen program Flex tarafından üretilen programlama birlikte derlenip link edilmektedir. Programcı tarafından (ya da Flex tarafından) üretilecek yylex fonksiyonun protoipi aşağıdaki gibi olmalıdır:

int yylex(void);

Ayrıca Bison tarafından üretilen C kodu bir gramer hatası ile karşılaşıldığında yyerror isimli bir fonksiyonu çağırmaktadır. Bu fonksiyonun tanımlanması da Bison programcısının sorumluluğundadır. yyerror fonksiyonunun prorotipi şöyledir:

void yyerror(const char \*s);

Özetle programcının Bison dosyasını derleyip link edebilmesi yylex ve yyerror isimli iki fonksiyonu barındırması gerekmektedir. Ancak bu yylex ve yyerror dosyalarının prototipleri de herhangi bir biçimde Bison’un ürettiği dosyalarda bulunmadığı için bu prototip bildirimlerinin Bison dosyasının yukarısında “Giriş” bölümünde yapılması gerekir.

Bison dosyasının “Gramer Kuralları” bölümüne yerleştirilecek BNF gramerinde uygulanacak kurallar şunlardır:

- Kuraların sol tarafı ara sembollerden oluşur. Ara sembollerden sonra bir tane ‘:’ atomu getirilir. Kuralların sağ tarafı ise ara semboller ve son sembollerden (atomlardan) oluşturulabilir. Örneğin:

A: B ‘+’ C;

Bu kuralda açılmak istenen ara sembol A’dır. Burada B ve C ara semboller, ‘+’ ise son semboldür.

- Kurallardaki boşluk karakterlerinin bir önemi yoktur. Örneğin kuralın sağ tarafında istenildiği kadar boşluklar bırakılabilir. (Boş satırlar da benzer olarak boşluk karakterleri olduğu için bulundurulabilir.) Örneğin:

A:

B ‘+’ B;

- Kuralların sonu noktalı virgül meta karakteri ile bitirilir.

- Eğer kuralın sol tarafındaki ara sembol için seçenek varsa bunlar farklı kurallar gibi de oluşturulabilir, aynı kuralda ‘|’ meta karakterleriyle de oluşturulabilir. Örneğin:

A: B ‘+’ B;

A: B ‘-‘ B;

Bu iki kural şöyle de yazılabilirdi:

A: B ‘+’ B

| B ‘-‘ B;

- Bison’da son semboller (atomlar) üç biçimde belirtilirler:

1) Tek tırnak içerisinde. Tek tırnağın içeisinde tek bir karakter bulunmak zorundadır. Bu durumda bu tek tırnağın içerisindeki karakter atomun türü olarak yylex fonksiyonundan elde edilen değerdir. Örneğin Bison ‘+’ gibi bir seöbol gördüğünde bu ‘+’ sembolünün ASCII tablosundaki sıra numarası 97 olduğu için bununla yylex fonksiyonun geri döndürdüğü 97 değerini eşleştirir.

2) Bir karakterden uzun olan atomlar %token direktifi ile dosyanın “Bison Bildirimleri” kısmında belirtilmelidirler. Bu durumda Bison bu direktifle bildirilen aotomun türüne ilişkin değeri kendisi belirlemektedir. Tabii yylex fonksiyonun bu atom için aynı değerle geri döndürülmesi gerekir. Örneğin:

%token SQRT

...

Expression:

SQRT ‘(‘ Expression ‘)’;

eğer istenirse atomun tür değeri de belirtilebilir. Örneğin:

%token SQRT 260

3) Atomlar okunabilirliği artırmak için iki tırnak içerisinde de belirtilebilirler. Ancak iki tırnak içerisinde belirtilen atomların yine %token direktifi ile aşağıdaki gibi bildirilmeleri gerekir:

%token SQRT “sqrt”

...

Expression:

“sqrt” ‘(‘ Expression ‘)’

Bu biçimde istenirse atom için tür numarası da verilebilir:

%token SQRT 260 “sqrt”

Eğer son semboller tek tırnak içerisinde belirtilmemiş fakat %token direktifi ile belirtilmişlerse bunları büyük harflerle isimlendirmek bir gelenektir.

- Bison’da kuralların sağ tarafına { ... } blokları içerisine istenidliği kadar C kodları yerleştirilebilir. Bu kodlara Bison manüelinde “kuralın yapılacak işlemler kısmı (action)” denilmektedir. Örneğin:

A:

B ‘+’ { printf(“test1\n”); } B { printf(“test\n”); };

Kurallardaki C kodları gramer çözümlenirken ilgili kurala uygun sağlandığı noktada çalıştırılmaktadır. Başka bir deyişle Bison bu kodları ilgili kuralla ilgili noktada uyuşum sağlandığında çağrılacak biçimde ürettiği C dosyasına yerleştirir.

Bison’da ikinci %% karakterinden sonra yine programcının C kodları bulundurulabilir. Asıl parse işini yapan fonksiyon yyparse isimli fonksiyondur. Bu nedenle main fonksiyonunda yyparse fonksiyonun çağrılması gerekir. Bison’da hataların değerlendirilmesi konusunun bazı ayrıntıları vardır. Bu ayrıntılar ileride ele alınacaktır.

**Bison İle Flex’in Birlikte Kullanılması**

Normal olarak Bison aracı Flex ile birlikte kullanılmaktadır. Bison sıradkai atomu yylex fonksiyonunu çağırarak elde eder. Bu fonksiyon da Flex tarafındna sağlanır. Anımsanacağı gibi yylex fonksiyonunun geri dönüş değeri atom türüdür. Bu durumda yylex’in geri döndürdüğü atom türleriyle Bison’daki türlerin sayısal bakımdan uyuşması gerekir. Bison’da eğer atom tek tırnak içerisinde verilmişse Flex’te de yylex aynı değerle geri döndürülerek uyum sağlanabilir. Ancak Bison’da atomlar %token direktifi ile belirtilmişse ve atoma değer verilmemişse bu durumda Bison atomun tür değerini kendisi tespit etmektedir. İşte Bison tüm %token direktifleriyle belirtilen atomlara ilişkin sembolik sabit bildirimlerini istenirse bir başlık dosyasında toplayabilmektedir. Bunun için bison “-d” seçeneğiyle çalıştırılır. Örneğin:

bison -d sample.y (Windows’ta win\_bison)

Burada ürün olarak default durumda duzantısı “.h” olan bir dosya elde edilir. Yukarıdaki örnekte “-o” seçeneği kullanılmadığı için üretilecek C dosyası “sample.tab.c” ve başlık dosyası da “sample.tab.h” biçiminde olacaktır. Bu durumda Bison ile Flex birlikte kullanılacaksa önce Bison dosyası işleme dokulup “.h” dosyası elde edilmeli bu dosya Flex’te include edilerek yylex fonksiyonun aynı aton tür değerleriyle geri dönüşü sağlanmalıdır.

Anımsanacağı gibi yylex atomu bulduğunda onun türü kodu ile geri dönmekte ancak bulduğu atomun yazısını da yytext adresiyle belirtilen diziye yerleştirmekteydi. Ancak Bison yylex fonksiyonu çağırdıktamn sonra atomun türünü elde edince onun değerini yytext değişkeninden değil yylval isimli bir değişkenden elde eder. yylval değişkeni YYSTYPE isimli bir tür türündendir. Bu tür default olarak int biçimde typedef edilmektedir. Ancak ileride de görüleceği üzere bu türün atomdan atoma ve kuraldan kurala değişebilmesi nedeniyle bir union olarak bildirilmesi gerekebilir. Bunun için Bison’da %union direktifi kullanılmaktadır. Eğer bu direktif kullanılırsa yylval bu direktife belirtilen union türünden olacaktır. Programcının Flex’te atomu bulunca eğer onun yalnızca türü değil değeri de önemliyle o değeri yylval içerisine yerleştirmesi gerekir. Çünkü Bison onu oradan almaktadır.

Atomların değerleri birbirlerinden farklı olabileceği için %token direktifinde onların türleri açısal parantezler içerisinde belirtilir. Bu belirtme %union direktifindeki değişken isimleri kullanılarak yapılır. Ancak başka seçenekler de vardır. Örneğin:

%union {

double val;

const char \*id;

};

%token NUMBER<val>

%token IDENTIFIER<id>

**Kurallarddaki Değerlerin Kullanılması**

Bison’da her kuralda ara sembollerden ve atomlardan elde edilen değerler vardır. Bu değerlerin türleri de söz konusudur. Atomların türleri yukarıda belirtildiği gibi açısal parantezlerle belirleniyordu. Ara sembollerin türleri de %type direktifleriyle belirlenmektedir. Örneğin:

%union {

double val;

const char \*id;

};

%token NUMBER<val>

%token IDENTIFIER<id>

%type Expression<val>

Burada Expression ara sembolünün türü val’ın türü olan double’dır.

Ara sembollerin değerleri blok içerisindeki C kodlarında $n sembolüyle kullanılabilmektedir. Buradaki n değeri kuralın sağ tarafındaki atom ya da ara sembolün pozisyon numarasıdır. Kuralın sol tarafındaki ara sembolün değeri $$ ile belirtilmektedir. Örneğin:

%union {

double val;

const char \*id;

};

%token NUMBER<val>

%token IDENTIFIER<id>

%type AdditiveExpression<val>

%type MultiplicativeExpression<val>

...

AdditiveExpression:

AdditiveExpression { $$ = $1; }

| AdditiveExpression ‘+’ MultiplicativeExpression { $$ = $1 + $3; }

| AdditiveExpression ‘-’ MultiplicativeExpression { $$ = $1 - $3; }

;

Yukarıda anlatılan bilgiler size bir örnek yapılmadan oldukça soyut ve anlamsız gelebilir. Ancak bu bilgiler bir örnek için en temel bilgilerdir. Yulkarıdaki açıklamalarda neyin ne anlama geldiği örneklerle yavaş yavaş anlaşılabilecektir.

**yyParse Fonksiyonun Çalışma Biçimi**

Bison’daki yyParse fonksiyonu tıpkı Flex’teki yylex fonksiyonu gibi birden fazla kez çağrılabilmektedir. yyParse her çağrıldığında kaynak dosyada kaldığı yerden devam eder. Programcı da isterse yyParse fonksiyonunu bir kez değil birden fazla kez çağırabilmektedir. Bir sentaks hatası olduğunda Bison yyerror fonksiyonunu çağırdıktan sonra yyParse fonksiyonunu sonlandırmaktadır. Bu durumda biz yyParse fonksiyonunu yeniden çağırarak işlemlerin devam etmesini sağlayabiliriz. Aslında bir hata olduğunda yyerror fonksiyonun çağrılması da engellenebilmektedir. Bu konu ileride ele alınacaktır.

**Örnek Bir Komut Satırı Hesap Makinesi**

Burada komut satırında her bir girişte bir ifadenin değerini hesaplayıp ekrana yazdıran bir uygulama yapacağız. Örneğin bu uygulama sayesinde:

3 \* (2 + 7) \* sqrt(100) + 2

gibi birşelmin sonucu ENTER tuşuna basıldığında aşağıda görüntülenecektir. “quit “ gibi bir komut verildiğinde de program sonlanacaktır.

Önce bu uygulama için atomlarına ayırma işlemini yapan bir Flex dosyası hazırlayalım. Aslında Bison’dan elde edilen atom değerlerinin Flex’e verilmesi gerekir. Ancak biz bunun yapıldığını varsayarak Bison’dan elde edilen dosyayı yukarıya include etmiş olalım. Basit bir hesap makinesi için Flex kodu şöyle olabilir:

/\* calc. l \*/

%option noyywrap

%{

#include <math.h>

#include "calc.tab.h"

%}

%%

[0-9]+ { yylval.val = atof(yytext); return NUMBER; }

[+\-\*()/\n] { return \*yytext; }

exit { return EXIT; }

[ \t]

. { return \*yytext; }

%%

Bu Flex kodunu açıklayalım. Kodun başında iki dosyanın include edildiğini görüyorsunuz. Kod içerisinde atof fonksiyonu kullanıldığı için <math.h> dosyası include edilmiştir. “calc.tab.h” dosyası bison tarafından üretilen dosyadır. Bu dosyanın içerisinde NUMBER sembolik sabitinin ve yylval değişkenin bildirimleri bulunur. yylval bizim Bison dosyamıza göre bir union türündendir. Bu union türünün val isimli bir elemanı vardır. Bison yylex fonksiyonu çağırdığında atomun türünün NUMBER olduğunu görünce onun değerinin yylval nesnesinin val elemanından alacaktır. Bu Flex dosyasında boşluk ve tab karakterlerinin geçildiğini görüyorsunuz. Boşlık ve tab karakterler elde edildiğinde yylex bir değere geri döndürülmemiştir. Dolayısıyla parser modülünün bu karakterlerin olduğundan bile haberi yoktur. ‘+’, ‘-‘, ‘\*’, ‘(‘, ‘)’, ‘/’ ve ‘\n’ karakterleri ayrı birer atom olarak değerlendirilmiş ve bunların ASC II karşılıkları atom türü olarak belirlenmiştir. (Köşeli parantez içerisinde \*, +, (, ) karakterleri meta karakterleri olarak değerlendirilmemektedir. Ancak ‘-‘ karakteri aralık belirttiğinden köşeli parantez içerisinde ters bölülenmiştir.)

Şimdi bu basit hesap makinesi için Bison dosyasını oluşturalım:

/\* calc.y \*/

%{

#include <stdio.h>

#include <stdlib.h>

int yylex(void);

void yyerror(const char \*str);

%}

%union {

double val;

};

%token <val> NUMBER

%token EXIT

%type <val> Expression

%type <val> AdditiveExpression

%type <val> MultiplicativeExpression

%type <val> UnaryExpression

%type <val> PrimaryExpression

%%

Calc:

Line Calc

|

;

Line:

Expression '\n' { printf("%f\n", $1); } Line

| '\n'

| EXIT { return 0; }

|

;

Expression:

AdditiveExpression { $$ = $1; }

;

AdditiveExpression:

MultiplicativeExpression { $$ = $1; }

| AdditiveExpression '+' MultiplicativeExpression {$$ = $1 + $3; }

| AdditiveExpression '-' MultiplicativeExpression {$$ = $1 - $3; }

;

MultiplicativeExpression:

UnaryExpression { $$ = $1; }

| MultiplicativeExpression '\*' UnaryExpression {$$ = $1 \* $3; }

| MultiplicativeExpression '/' UnaryExpression {$$ = $1 / $3; }

;

UnaryExpression:

PrimaryExpression { $$ = $1; }

| '-' UnaryExpression { $$ = -$2; }

| '+' UnaryExpression { $$ = +$2; }

;

PrimaryExpression:

NUMBER { $$ = $1; }

| '(' Expression ')' { $$ = $2; }

;

%%

int main(void)

{

yyparse();

return 0;

}

void yyerror(const char \*str)

{

fprintf(stderr, "syntax error!\n");

}

Kodun “Giriş (Prologue)” kısmında gerekli C başlık dosyalarının include edildiğini ve ayrıca yylex ve yyerror fonksiyonlarının prototip bildirimlerinin yapıldığını görüyorsunuz:

%{

#include <stdio.h>

#include <stdlib.h>

int yylex(void);

void yyerror(const char \*str);

%}

Bison yylex ve yyeror fonksiyonlarını kendi ürettiği kodda kullanmaktadır. Ancak bunların prototip bildirimlerini kendisi yapmamaktadır. Bu nedenle bizim dosyanın “Giriş” kısmında bu bildirimleri yapmamız gerekir.

Dosyada “Giriş” kısmından sonra “Bison Bildirimleri” kısmında aşağıdaki bildirimler yapılmıştır:

%union {

double val;

};

%token <val> NUMBER

%token EXIT

%type <val> Expression

%type <val> AdditiveExpression

%type <val> MultiplicativeExpression

%type <val> UnaryExpression

%type <val> PrimaryExpression

Buradaki %union bildirimi yylval isimli nesnenin türünü belirtmektedir. Eüer bu bildirim yapılmazsa yylval default olarak int kabul edilir. %union bildirimi yylval nesnesinin burada belirtilen C tarzı union türünden olduğunu belirtir. yylval aslında Bison tarafından bizim aracılığımızla kullanılır. Biz Bison kodunda kurallarda $n ifadesiyle bunları kullanırız. Benzer biçimde Bison’da her ara sembolün de bir türü vardır. Bu tür de yine bizim tarafımızdan $n ifadeleriyle kullanılır. İşte %type direktifi ara sembollerin türünü belirtmektedir. Ara semboller de yine yylval nesnesi yoluyla bize verilmektedir.

%Token ve %type direktiflerindeki tür açısal parantezler içerisinde dolaylı olarak belirtilmektedir. Eğer bu direktiflerde açısal parantezler kullanılmazsa default tür int olarak kabul edilir. Açısal parantezlerin içerisinde union bildirimindeki eleman isimleri yazılır. Bu durumda bu elemanın türü (örneğimde val) o atomun ya da ara sembolün türü olur.

Hesap mekinesi uygulamasında “Kurallar” bölünmü aşağıdaki gibi oluşturulmuştur:

Calc:

Line Calc

|

;

Line:

Expression '\n' { printf("%f\n", $1); } Line

| '\n'

| EXIT { return 0; }

|

;

Expression:

AdditiveExpression { $$ = $1; }

;

AdditiveExpression:

MultiplicativeExpression { $$ = $1; }

| AdditiveExpression '+' MultiplicativeExpression {$$ = $1 + $3; }

| AdditiveExpression '-' MultiplicativeExpression {$$ = $1 - $3; }

;

MultiplicativeExpression:

UnaryExpression { $$ = $1; }

| MultiplicativeExpression '\*' UnaryExpression {$$ = $1 \* $3; }

| MultiplicativeExpression '/' UnaryExpression {$$ = $1 / $3; }

;

UnaryExpression:

PrimaryExpression { $$ = $1; }

| '-' UnaryExpression { $$ = -$2; }

| '+' UnaryExpression { $$ = +$2; }

;

PrimaryExpression:

NUMBER { $$ = $1; }

| '(' Expression ')' { $$ = $2; }

;

Burada “Calc” kuralı “Line”lardan, “Line”lar da “Expression” ve \n’ atomlarından oluşmaktadır. Expressipn kuralları klasik C’nin gramer kurallarına benzetilerek oluşturulmuştur. Biz burada kuralların “Yapılacak Eylem (Action)” kısımları üzerinde duracağız. Her kural belirlendiğinde Bison o noktada küme parantezleri içeriisndeki C kodlarının çağrılmasını sağlar. Küme parantezleri aslında kuralların her yerine yerleştirilebilir. Yani onların en sağında bulunmak zorunda değildir. Küme parantezleri içerisindeki C kodlarında $$ meta karakterleri o kuralın sol tarafındaki ara sembole atanacak değeri, $n meta karakterleri ise ara sembolün sağ tarafındaki ara sembol ya da atomun değerini belirtmektedir. Bison’un aşağıdan yukarıya (Bottom-Up) bir parser algoritması uyguladığını anımsayınız. Bu durumda ifade içerisindeki atomlar birleştirile birleştirile yukarıya doğru çıkılacak ve en yukara Expression tek bir değer olarak elde edilecektir.

Şimdi yukarıdaki hesap makinesini biraz daha geliştirelim. Örneğin ona çeşitli “built-in” fonksiyonlar ekleyelim:

/\* calc.l \*/

%option noyywrap

%{

#include <math.h>

#include "calc.tab.h"

%}

%%

[0-9]+ { yylval.val = atof(yytext); return NUMBER; }

[+\-\*()/\n!%] { return \*yytext; }

exit { return EXIT; }

sqrt { return SQRT; }

pow { return POW; }

[ \t]

. { return \*yytext; }

%%

/\* calc.y \*/

%{

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

int yylex(void);

void yyerror(const char \*str);

double factorial(double n);

%}

%union {

double val;

};

%token <val> NUMBER

%token SQRT

%token EXIT

%token POW

%type <val> Expression

%type <val> AdditiveExpression

%type <val> MultiplicativeExpression

%type <val> UnaryExpression

%type <val> PostfixExpression

%type <val> PrimaryExpression

%%

Calc:

Line Calc

|

;

Line:

Expression '\n' { printf("%f\n", $1); } Line

| '\n'

| EXIT { return 0; }

|

;

Expression:

AdditiveExpression { $$ = $1; }

;

AdditiveExpression:

MultiplicativeExpression { $$ = $1; }

| AdditiveExpression '+' MultiplicativeExpression {$$ = $1 + $3; }

| AdditiveExpression '-' MultiplicativeExpression {$$ = $1 - $3; }

;

MultiplicativeExpression:

UnaryExpression { $$ = $1; }

| MultiplicativeExpression '\*' UnaryExpression {$$ = $1 \* $3; }

| MultiplicativeExpression '/' UnaryExpression {$$ = $1 / $3; }

| MultiplicativeExpression '%' UnaryExpression {$$ = (int)$1 % (int)$3; }

;

UnaryExpression:

PostfixExpression { $$ = $1; }

| '-' UnaryExpression { $$ = -$2; }

| '+' UnaryExpression { $$ = +$2; }

;

PostfixExpression:

PrimaryExpression { $$ = $1;}

| SQRT '(' Expression ')' { $$ = sqrt($3); }

| POW '(' Expression ',' Expression ')' { $$ = pow($3, $5); }

| PostfixExpression '!' { $$ = factorial($1); }

;

PrimaryExpression:

NUMBER { $$ = $1; }

| '(' Expression ')' { $$ = $2; }

;

%%

int main(void)

{

yyparse();

return 0;

}

double factorial(double n)

{

int i;

double fact = 1;

for (i = 1; i <= n; ++i)

fact \*= i;

return fact;

}

void yyerror(const char \*str)

{

fprintf(stderr, "syntax error!\n");

}

Burada gramere % operatörünün, ! (faktöryel) operatörünün ve sqrt ve pow fonksiyonlarının eklendiğine dikkat ediniz.



Yuklarıdaki örneği Windows’ta şöyle derleyip bağlayabilirsiniz:

win\_flex --wincompat calc.l

win\_bison -d calc.y

cl /Fe:calc.exe lex.yy.c calc.tab.c

Linux sistemlerinde de aynı işlem şöyle yapılabilir:

flex calc.l

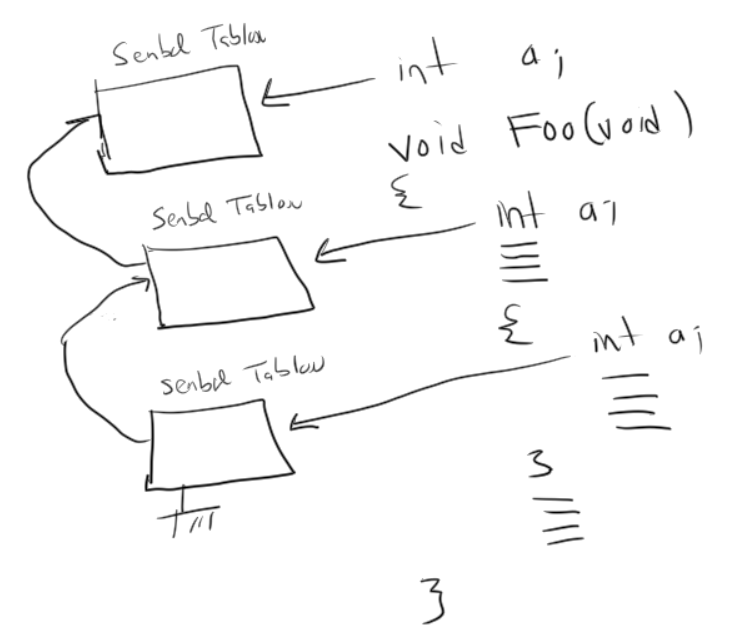
bison -d calc.y

gcc -o calc lex.yy.c calc.tab.c

**Sembol Tabloları (Symbol Tables)**

Derleyiciler ve yorumlayıcılar değişkenleri (identifiers) ilk gördüklerinde (dillerin çoğunda bunlar ilk kez bildirim yoluyla görülürler) bunları sembol tablosu denilen bir tabloya yerleştirirler. Sembol tablolarında değişkenlerin isimleri, türleri, faaliyet alanları, deposal özellikleri (örneğin static gibi), değiştirilebilirliği (const’luğu), kaynak koddaki bildirim yerleri gibi bilgiler tutulmaktadır. Yorumlayıcılarda değişkenlerin değerleri de yine sembol tablolarında tutulabilmektedir. Sembol tabloları genellikle hash tabloları biçiminde gerçekleştirilir. Değişkenlerin adı da anahtar olarak kullanılır. Tabii sembol tablolarının içeriği dilden dile o dile bağlı olarak değişebilmektedir. Sembol tabloları ile string tablolarını karıştırmamak gerekir. String tablolarından amaç bir değişkenin isminin yalnızca bir kez depolanmasını sağlamaktır. Halbukşi sembol tablolarında değişkenlere ilişkin tüm bilgiler tutulmaktadır.

C gibi iç içe faaliyet alanlarının bulunduğu dillerde farklı faaliyet alanlarında aynı isimli değişkenler bildirilebilmektedir. Değişken ismi sembol tablosuna anahtar yapıldığına göre ve anahtarın tek olması gerektiğine göre sembol tabloları faaliyet alanlarına göre nasıl gerçekleştirilecektir? İşte genellikle bu tür durumlarda her faaliyet alanı için ayrı bir sembol tablosu oluşturulmaktadır. Sembol tabloları da içten dışarıya doğru bağlı liste biçiminde birarada tutulur.



Böylece değişken önce alttaki sembol tablosunda aranır, orada bulunursa yukarıya çıkılmaz. Bulunamazsa yukarıya çıkılarak ilerlenir.

Pekiyi en basit bir sembol tablosu gerçekleştirimi nasıl olabilir? eğer biz değişken isimlerini bir karakterle kısıtlarsak o karakterin karakter tablosundaki sıra numarasını bir hash değeri olarak kullanabiliriz. Örneğin dilimizde yalnızca küçük harflerden değişken isimleri oluşturulabilsin. Değişkenlerin bilgileri de ID\_INFO isimli bir yapıyla temsil edilsin. Bu durumda sembol tablomuz şöyle olabilir:

struct ID\_INFO symbolTable[26];

Tabii değişken isimleri uzun olabilecekse mecburen hash tablosu kullanmak gerekir. Şimdi daha önce yamış olduğumuz hesap makinesine değişken ekleyelim. Sembol tablosunda yalnızca değişkenin değeri bulunuyor olsun:

/\* calc.l \*/

%option noyywrap

%{

#include <math.h>

#include "calc.tab.h"

%}

%%

[a-z] { yylval.id = \*yytext; return IDENTIFIER; }

[0-9]+ { yylval.val = atof(yytext); return NUMBER; }

[+\-\*()/\n!%=] { return \*yytext; }

exit { return EXIT; }

sqrt { return SQRT; }

pow { return POW; }

[ \t]

. { return \*yytext; }

%%

/\* calc.l \*/

%option noyywrap

%{

#include <math.h>

#include "calc.tab.h"

%}

%%

[a-z] { yylval.id = \*yytext; return IDENTIFIER; }

[0-9]+ { yylval.val = atof(yytext); return NUMBER; }

[+\-\*()/\n!%=] { return \*yytext; }

exit { return EXIT; }

sqrt { return SQRT; }

pow { return POW; }

[ \t]

. { return \*yytext; }

%%

Windows’ta derleme işlemi aynı biçimde yapılabilir:

win\_flex --wincompat calc.l

win\_bison -d calc.y

cl /Fe:calc.exe lex.yy.c calc.tab.c

Linux ve Mac OS X sistemlerinde de şöyle yapılabilir:

flex calc.l

bison -d calc.y

gcc -o calc lex.yy.c calc.tab.c

Tabii normal olan durum sembol tablolarının hash tablosu veri yapısıyla gerçekleştirilmesidir. Şimdi hash tablosu kullanarak değişken isimlerini tek karakter olmaktan çıkartalım:

/\* stringtable.h \*/

#ifndef STRINGTABLE\_H\_

#define STRINGTABLE\_H\_

#include <stddef.h>

/\* Symbolic Constants \*/

#define TABLE\_SIZE 1000

/\* Type Declarations \*/

typedef struct tagSTRNODE {

char \*str;

size\_t size;

struct tagSTRNODE \*next;

} STRNODE;

/\* Function Prototypes \*/

char \*lookup\_str\_size(const char \*str, size\_t size);

char \*lookup\_str(const char \*str);

void destroy\_string\_table(void);

#endif

/\* stringtable.c \*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "stringtable.h"

/\* static Function prtotypes \*/

static size\_t hashFunc(const char \*str, size\_t size);

/\* Global Data Definitions \*/

STRNODE \*g\_hashTable[TABLE\_SIZE];

/\* Function Definitions \*/

static size\_t hashFunc(const char \*str, size\_t size)

{

unsigned long hash = 5381;

int i;

for (i = 0; i < size; ++i)

hash = ((hash << 5) + hash) + str[i];

return hash % TABLE\_SIZE;

}

char \*lookup\_str\_size(const char \*str, size\_t size)

{

size\_t hash;

STRNODE \*node;

hash = hashFunc(str, size);

node = g\_hashTable[hash];

while (node != NULL) {

if (size == node->size && !strncmp(str, node->str, size))

return node->str;

node = node->next;

}

if ((node = (STRNODE \*)malloc(sizeof(STRNODE))) == NULL)

return NULL;

if ((node->str = (char \*)malloc(size + 1)) == NULL) {

free(node);

return NULL;

}

strncpy(node->str, str, size + 1);

node->size = size;

node->next = g\_hashTable[hash];

g\_hashTable[hash] = node;

return node->str;

}

char \*lookup\_str(const char \*str)

{

char \*end = str;

while (\*end != '\0')

++end;

return lookup\_str\_size(str, (size\_t)(end - str));

}

void destroy\_string\_table(void)

{

int i;

STRNODE \*node, \*tempNode;

for (i = 0; i < TABLE\_SIZE; ++i) {

node = g\_hashTable[i];

while (node != NULL) {

tempNode = node->next;

free(node->str);

free(node);

node = tempNode;

}

}

}

/\* calc.l \*/

%option noyywrap

%{

#include <math.h>

#include "calc.tab.h"

#include "stringtable.h"

%}

%%

exit { return EXIT; }

sqrt { return SQRT; }

pow { return POW; }

[a-z][a-z0-9]\* { yylval.id = lookup\_str(yytext); return IDENTIFIER; }

[0-9]+ { yylval.val = atof(yytext); return NUMBER; }

[+\-\*()/\n!%=] { return \*yytext; }

[ \t]

. { return \*yytext; }

%%

/\* calc.y \*/

%{

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include "stringtable.h"

/\* Symbolic Constants \*/

#define TABLE\_SIZE 1000

/\* Type Declarations \*/

typedef struct tagID\_INFO {

char \*id;

double val;

struct tagID\_INFO \*next;

} ID\_INFO;

/\* Function Prototypes \*/

int yylex(void);

void yyerror(const char \*str);

double factorial(double n);

double get\_identifier\_value(const char \*id);

double set\_identifier\_value(const char \*id, double val);

static size\_t hashFunc(const char \*str);

void destroy\_symbol\_table(void);

/\* Global Definitions \*/

ID\_INFO \*g\_symbolTable[TABLE\_SIZE];

%}

%union {

double val;

const char \*id;

};

%token <val> NUMBER

%token <id> IDENTIFIER

%token SQRT

%token EXIT

%token POW

%type <val> AssignmentExpression

%type <val> Expression

%type <val> AdditiveExpression

%type <val> MultiplicativeExpression

%type <val> UnaryExpression

%type <val> PostfixExpression

%type <val> PrimaryExpression

%%

Calc:

Line Calc

|

;

Line:

Expression '\n' { printf("%f\n", $1); } Line

| '\n'

| EXIT { return 0; }

|

;

Expression:

AssignmentExpression { $$ = $1; }

;

AssignmentExpression:

AdditiveExpression { $$ = $1; }

| IDENTIFIER '=' AdditiveExpression { $$ = set\_identifier\_value($1, $3); }

;

AdditiveExpression:

MultiplicativeExpression { $$ = $1; }

| AdditiveExpression '+' MultiplicativeExpression {$$ = $1 + $3; }

| AdditiveExpression '-' MultiplicativeExpression {$$ = $1 - $3; }

;

MultiplicativeExpression:

UnaryExpression { $$ = $1; }

| MultiplicativeExpression '\*' UnaryExpression {$$ = $1 \* $3; }

| MultiplicativeExpression '/' UnaryExpression {$$ = $1 / $3; }

| MultiplicativeExpression '%' UnaryExpression {$$ = (int)$1 % (int)$3; }

;

UnaryExpression:

PostfixExpression { $$ = $1; }

| '-' UnaryExpression { $$ = -$2; }

| '+' UnaryExpression { $$ = +$2; }

;

PostfixExpression:

PrimaryExpression { $$ = $1;}

| SQRT '(' Expression ')' { $$ = sqrt($3); }

| POW '(' Expression ',' Expression ')' { $$ = pow($3, $5); }

| PostfixExpression '!' { $$ = factorial($1); }

;

PrimaryExpression:

NUMBER { $$ = $1; }

| IDENTIFIER { $$ = get\_identifier\_value($1); }

| '(' Expression ')' { $$ = $2; }

;

%%

int main(void)

{

while (yyparse() != 0)

;

destroy\_string\_table();

destroy\_symbol\_table();

return 0;

}

double factorial(double n)

{

int i;

double fact = 1;

for (i = 1; i <= n; ++i)

fact \*= i;

return fact;

}

double get\_identifier\_value(const char \*id)

{

ID\_INFO \*idInfo;

size\_t hash;

hash = hashFunc(id);

idInfo = g\_symbolTable[hash];

while (idInfo != NULL) {

if (id == idInfo->id)

return idInfo->val;

idInfo = idInfo->next;

}

fprintf(stderr, "warning: variable not found: '%s', zero value used instead\n", id);

return 0;

}

double set\_identifier\_value(const char \*id, double val)

{

ID\_INFO \*idInfo;

size\_t hash;

hash = hashFunc(id);

idInfo = g\_symbolTable[hash];

while (idInfo != NULL) {

if (id == idInfo->id) {

idInfo->val = val;

return val;

}

idInfo = idInfo->next;

}

if ((idInfo = (ID\_INFO \*)malloc(sizeof(ID\_INFO))) == NULL) {

fprintf(stderr, "cannot allocate memory!\n");

exit(EXIT\_FAILURE);

}

idInfo->id = id;

idInfo->val = val;

idInfo->next = g\_symbolTable[hash];

g\_symbolTable[hash] = idInfo;

return val;

}

void destroy\_symbol\_table(void)

{

int i;

ID\_INFO \*idInfo, \*tempIdInfo;

for (i = 0; i < TABLE\_SIZE; ++i) {

idInfo = g\_symbolTable[i];

while (idInfo != NULL) {

tempIdInfo = idInfo->next;

free(idInfo);

idInfo= tempIdInfo;

}

}

}

static size\_t hashFunc(const char \*str)

{

unsigned long hash = 5381;

int i;

for (i = 0; str[i] != '\0'; ++i)

hash = ((hash << 5) + hash) + str[i];

return hash % TABLE\_SIZE;

}

void yyerror(const char \*str)

{

fprintf(stderr, "syntax error!\n");

}

Burada string tablosunun da kullanıldığına dikkat ediniz. Sembol (değişken ismi) lexical analiz modülü tarafından bulunduğunda string tablosuna yerleştirilmiştir. Parser modülü string tablosundaki bu isme referans etmektedir.

Şimdi de yukarıdaki makinesi örneğini basit bir yorumlayıcı haline getirelim. Yani Flex girişleri klavyeden (stdin) değil, bizim belirlediğimiz bir dosyadan alsın. Komutları peşi sıra çalıştırsın. print isimli isimli özel bir komut da ifadenin değerini ekrana yazdırsın. Bunun dışında ifadelerin sonuçları ekrana yazdırılmasın.

**Bison Parser’ında Kurallar Arası Geçişler**

Anımsanacağı gibi Bison aşağıdan yukarıya LALR(1) türü bir parser algortiması kullanmaktadır. Aşağıdan yukarıya parse işlemi parse ağacının oluşturulmasını kolaylaştırmaktadır. Çünkü bu yöntemde önce alt düğümler sonra üst düğümler elde edilmektedir. Burada küçük bir örnekle Bison’un aşağıdan yukarıya parse işlemi sırasında kurallardan nasıl geçildiği ele alınacaktır. Örneğin aşağıdaki gibi basit bir gramer olsun:

/\* test.y \*/

%{

#include <stdio.h>

/\* Function Prototypes \*/

int yylex(void);

void yyerror(const char \*str);

%}

%token NUMBER

%token IDENTIFIER

%type <val> AssignmentExpression

%type <val> Expression

%type <val> AdditiveExpression

%type <val> MultiplicativeExpression

%type <val> PrimaryExpression

%union {

int val;

}

%%

Expression:

AssignmentExpression '\n' { printf("AssignmentExpression\n"); }

|

;

AssignmentExpression:

AdditiveExpression { printf("AdditiveExpression\n"); }

| IDENTIFIER '=' AdditiveExpression { printf("IDENTIFIER '=' AdditiveExpression\n");}

;

AdditiveExpression:

MultiplicativeExpression { printf("MultiplicativeExpression\n"); }

| AdditiveExpression '+' MultiplicativeExpression { printf("AdditiveExpression '+' MultiplicativeExpression\n"); }

| AdditiveExpression '-' MultiplicativeExpression { printf("AdditiveExpression '-' MultiplicativeExpression\n"); }

;

MultiplicativeExpression:

PrimaryExpression { printf("PrimaryExpression\n"); }

| MultiplicativeExpression '\*' PrimaryExpression { printf("MultiplicativeExpression '\*' PrimaryExpression\n"); }

| MultiplicativeExpression '/' PrimaryExpression { printf(" MultiplicativeExpression '/' PrimaryExpression\n"); }

| MultiplicativeExpression '%' PrimaryExpression { printf("MultiplicativeExpression '%%' PrimaryExpression\n"); }

;

PrimaryExpression:

NUMBER { printf("NUMBER\n"); }

| IDENTIFIER { printf("IDENTIFIER\n"); }

| '(' Expression ')' { printf("'(' Expression ')'\n"); }

;

%%

int main(int argc, char \*argv[])

{

yyparse();

printf("Ok\n");

return 0;

}

void yyerror(const char \*str)

{

fprintf(stderr, "syntax error!\n");

}

Aşağıdaki gibi bir giriş yapıldığını düşünelim:

a + b

Bison’un aşağıdan yukarıya parse işlemi sırasında kurallar arası geçişler şöyle olacaktır:

IDENTIFIER

PrimaryExpression

MultiplicativeExpression

IDENTIFIER

PrimaryExpression

AdditiveExpression '+' MultiplicativeExpression

AdditiveExpression

AssignmentExpression

Burada Bison önce a’yı bulacak ve onun IDENTIFIER olduğunu belirleyecektir. Sonra ‘+’ çektiğinde bunun artık bunun Additive ‘+’ Multiplicative kuralı olabileceğini düşünür. Sonra b’yi çektiğinde bunun da bir IDENTIFIER olduğunu anlar ve bu kuralı oluşturarak yukarıya doğru gider. Şimdi de aşağıdaki gibi bir giriş yuapılmış olduğunu varsayalım:

a = 10 + 20

Burada Bison önce a’yı görecek bunun IDENTIFIER olduğunu belirleyecek sonra ‘=’ atomunu gördüğünde artık bunun IDENTIFIER ‘=’ AdditiveExpression kuralına uygun olabileceğini anlar. Tabii henüz “reduce” işlemini yapmaz. Sonra 10 atomunu çeker sonra ‘+’ atomunu ve sonra 20’yi çekerek grameri tamamlar. Ekranda şınlar görünecektir:

NUMBER

PrimaryExpression

MultiplicativeExpression

NUMBER

PrimaryExpression

AdditiveExpression '+' MultiplicativeExpression

IDENTIFIER '=' AdditiveExpression

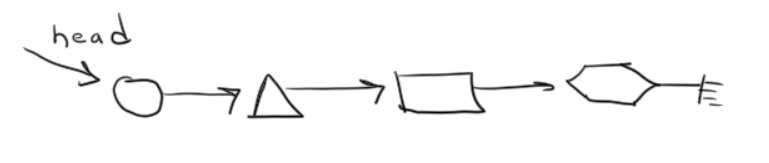
AssignmentExpression

Görldüğü gibi Bison’un aşağıdan yukarıya parse işlemi yapması programcının bir parse ağacı oluşturmasını kolaylaştırmaktadır.

**Heterojen Düğümlere Sahip Bağlı Listeler ve Ağaçlar Nasıl Oluşturulur?**

Soyut Sentaks Ağacı (Abstract Syntax Tree) oluşturulurken pek çok bağlı liste ve ağaç yapısının kullanılması gerekmektedir. Ancak bu bağlı listelerdeki ve ağaçlardaki düğümler aynı elemanlardan oluşan homojen düğümler olmayabilmektedir. İşte biz de bu bölümde heterojen düğümlerden oluşan bağlı listelerin nasıl ifade edilebileceği üzer

inde duracağız. Örneğin bir tek bağlı liste düşünelim. Fakat onların gösterdiği yerdeki düğümlerin içerikleri farklı olsun:



Buradaki şekiller farklı türdeki düğümleri temsil etmektedir. İşte heterojen düğümler sahip bağlı listeleri ya da ağaçları temsil etmek için birkaç yöntem kullanılabilmektedir. Bu yöntemlerin hepsinin bazı avantajları ve dezavantajları söz konusu olabilmektedir.

**1. Yöntem : Birliklerden Faydalanarak Tür Farklılığını Ortadan Kaldırmak**

Bu yöntemde düğümler sanki aynı türdenmiş gibi ele alınır. Ortak elemanlar yapının başına yerleştirilir. Farklı elemanlar da bir birlik içerisinde ifade edilir. Tabii ortak elemanların içerisine birliğin hangi elemanının o düğümde bulundurulduğunu belirlemek için bir type elemanı yerleştirilir. Örneğin tek bağlı liste için bu uygulama aşağıdaki gibi yapılabilir:

enum TYPES {

XNODE, YNODE, ZNODE

};

struct NODE {

int type;

struct NODE \*next;

union {

struct XNODE xnode;

struct YNODE ynode;

struct ZNODE znode;

} u;

};

Burada artık bir düğümn adresi elde edildiğinde onun içeriğine onun type elemanı yoluyla aşağıdaki gibi erişilir:

switch (node->type) {

case XNODE:

/\* node->u.xnode \*/

break;

case YNODE:

/\* node->u.ynode \*/

break;

case ZNODE:

/\* node->u.znode \*/

break;

}

Bu yöntemin avantajı basit olmasıdır. Ancak birlik içerisindeki elemanların türleri farklı olduğu için toplam kullanılan bellek miktarı büyük olacaktır. (Anımsanacağı gibi bir birlik için o birliğin en büyük elemanı kadar yer ayrılmaktadır.)

**2. Yöntem : Farklı Türden Düğümlerin Ortak Kısımlarının Aynı Türle İfade Edilmesi**

Bu yöntemde farklı türdeki düğüm yapılarının başındaki belli kısımları ortak bir yapıyla temsil edilir. Yine bu ortak elemanlar içeirisinde sonraki düğümü gösteren göstericiler ve düğümün türünü belirten type elemanı tutulur. Örneğin tek bağlı liste için bu yöntem şöyle uygulanabilir:

struct NODE {

int type;

struct NODE \*next;

};

struct XNODE {

int type;

struct NODE \*next;

....

};

struct YNODE {

int type;

struct NODE \*next;

....

};

struct ZNODE {

int type;

struct NODE \*next;

....

};

Burada XNODE, YNODE ve ZNODE yapılarının başlangıç kısımları aynıdır. Bu başlangıç kısımları NODE yapısındaki gibidir. O halde bu farklı yapıların adresleri tür dönüştürmesi ile sanki NODE türünden yapıları gösteriyormuş gibi organizasyon yapılabilir. Böylece biz kod içerisinde her düğümü NODE yapısı türündenmiş gibi değerlendiririz. Ancak bunların type kısmına bakarak tür dönüştürmesi ile gerçek türleri elde edebiliriz. Örneğin:

switch (node->type) {

case XNODE:

{

struct XNODE \*xnode = (struct XNODE \*)node;

...

}

break;

case YNODE:

{

struct YNODE \*ynode = (struct YNODE \*)node;

...

}

break;

case ZNODE:

{

struct ZNODE \*znode = (struct ZNODE \*)node;

...

}

break;

}

Ancak bu yöntemde C standartları bakımından küçük bir sorun vardır. C standartlarında yapının ilk elemanının adresinin yapı nesnesinin bütünsel adresiyle aynı olacağı belirtilmiştir. Ancak yapının sonraki elemanlarının artan adreslerde olduğu belirtilmişse de hizalama yüzünden aralarda boşlukların bırakılabilmesine olanak sağlanmıştır. Gerçi derleyiciler default durumda tüm yapılar için aynı hizalama biçimini kullanıyor olsalar da standartlar bağlamında bu durum garanti edilmemiştir. O halde yukarıdaki yapıyı standartlara uydurmak için ortak elemanların tek bir yapı elemanıyla ifade edilmesi gerekir:

struct NODE {

int type;

struct NODE \*next;

};

struct XNODE {

struct NODE node;

....

};

struct YNODE {

struct NODE node;

....

};

struct ZNODE {

struct NODE node;

....

};

Aslında bu yöntem nesne yönelimli programlama tekniğindeki türetme işlemine benzetilebilir. Çünkü orada da zaten nesnenin taban sınıf kısmı nesnenin yukarısında tutulmaktadır ve türemiş sınıftan taban sınıfa adres dönüştürmesi vardır.

Bu yöntemde farklı düğümler yalnızca kendileri kadar yer kaplarlar. Ancak gerçekleştirilmeleri biraz daha zahmetlidir.

**3. Yöntem : Farklı Türden Düğümlerin Elemanlarını Ayrı Bir Biçimde Tahsis Edip Onların Adreslerini Düğümde Tutmak**

Bu yöntemde yine tüm düğümler sanki aynı türdenmiş gibi aynı yapıyla temsil edilir. Ancak bunların bir gösterici elemanı vardır ve o eleman heterojen türe ilişkin bilgileri tutan yapıyı gösterir. Örneğin:

struct NODE {

int type;

struct NODE \*next;

void \*data;

};

...

switch (node->type) {

case XNODE:

{

struct XNODE \*xnode = (struct XNODE \*)node->data;

...

}

break;

case YNODE:

{

struct YNODE \*ynode = (struct YNODE \*)node->data;

...

}

break;

case ZNODE:

{

struct ZNODE \*znode = (struct ZNODE \*)node->data;

...

}

break;

}

Bu yöntemde her düğüm için iki ayrı dinamik tahsisat gerekmektedir.

**Tasarlanan Küçük Bir Dil İçin Bir Yorumlayıcının Gerçekleştirilmesi**

Bu bölümde küçük bir dil tasarlayıp o dil için bir yorumlayıcı yazacağız. Ancak yorumlayıcı yazımı için soyut bir sentaks ağacının da kurulması gerekmektedir. Bu sentaks ağacı heterojen düğümlerden oluşan bir ağaç biçimindedir. Öncelikle dili tasarlamakla işe başlayalım.

**Örnek Dilin Tasarımı**

Örnek dil prosedürel programlamayı destekleyen, içerisinde fonksiyonların bulunduğu, if gibi while ve for gibi kontrol değimlerinin bulunduğu bir dil olsun. Dildeiki veri türü bulunacaktır. Bunlar int ve double türleridir. void anahtar sözcüğü fonksiyonların geri dönüş değerlerinde kullanılabilir. Bu durum fonksiyonun bir değer geri döndürmeyeceği anlamına gelir. Programın başlangıç noktası main isimli fonksiyondur. Dilde iç içe bloklar bulunabilir. Bildirimler blokların başlarında yapılmak zorundadır. Bu dile bu dokümanlarda CSD dili denilecektir. CSD diline ilişkin örnek bir program şöyle olabilir:

int g\_x;

void Foo(int n)

{

int i;

i = 0;

while (i < n) {

print i;

++i;

}

}

void main()

{

Foo(10);

Foo(20);

}

**CSD Dilinin Grameri**

CSD Dilini atomlarına ayıran Flex kodu aşağıdaki gibidir:

/\* csd.l \*/

%option noyywrap

%{

#include "csd.tab.h"

#include "stringtable.h"

%}

%%

"int" { return INT; }

"double" { return DOUBLE; }

"void" { return VOID; }

"if" { return IF; }

"else" { return ELSE; }

"while" { return WHILE; }

"for" { return FOR; }

"do" { return DO; }

"return" { return RETURN; }

"++" { return PLUSPLUS; }

"--" { return MINUSMINUS; }

"==" { return EQUALITY; }

"!=" { return INEQUALITY; }

"<=" { return LESSOREQU; }

">=" { return GREOREQU; }

[a-z][a-z0-9]\* { yylval.id = lookup\_str(yytext); return IDENTIFIER; }

[0-9]+ { yylval.numberInt = atoi(yytext); return NUMBER\_INT; }

[0-9]+\.[0-9]+ { yylval.numberDouble = atof(yytext); return NUMBER\_DOUBLE; }

[+\-\*()/%=;{}] { return \*yytext; }

[ \t\n]

. { return \*yytext; }

%%

CSD dilinin Bison Grameri de aşağıdaki gibi oluşturulmuştur:

/\* csd.y \*/

%expect 1

%{

#include "csd.h"

%}

%union {

void \*astNode;

char \*id;

int numberInt;

double numberDouble;

};

%token INT

%token DOUBLE

%token VOID

%token IDENTIFIER

%token NUMBER\_INT

%token NUMBER\_DOUBLE

%token IF

%token ELSE

%token WHILE

%token FOR

%token DO

%token EQUALITY

%token INEQUALITY

%token GREOREQU

%token LESSOREQU

%token PLUSPLUS

%token MINUSMINUS

%token RETURN

%type <astNode> CSD

%type <astNode> Definitions

%type <astNode> Definition

%type <astNode> VariableDefinition

%type <astNode> FunctionDefinition

%type <astNode> TypeSpecifier

%type <astNode> VariableList

%type <astNode> ParameterList

%type <astNode> ArgumentList

%type <astNode> Argument

%type <astNode> Parameter

%type <astNode> Statements

%type <astNode> Statement

%type <astNode> WhileStatement

%type <astNode> DefinitionStatement

%type <astNode> IfStatement

%type <astNode> ForStatement

%type <astNode> SimpleStatement

%type <astNode> CompoundStatement

%type <astNode> ReturnStatement

%type <astNode> Expression

%type <astNode> ForExpression

%type <astNode> AssignmentExpression

%type <astNode> AdditiveExpression

%type <astNode> MultiplicativeExpression

%type <astNode> UnaryExpression

%type <astNode> PostfixExpression

%type <astNode> PrimaryExpression

%%

CSD:

Definitions

|

;

Definitions:

Definition

| Definitions Definition

;

Definition:

VariableDefinition

| FunctionDefinition

;

FunctionDefinition:

TypeSpecifier IDENTIFIER '(' ParameterList ')' CompoundStatement

;

VariableDefinition:

TypeSpecifier VariableList ';'

;

ParameterList:

Parameter

| ParameterList',' Parameter

|

;

Parameter:

TypeSpecifier IDENTIFIER

;

ArgumentList:

Argument

| ArgumentList ',' Argument

|

;

Argument:

Expression

;

VariableList:

IDENTIFIER

| VariableList ',' IDENTIFIER

;

TypeSpecifier:

INT

| DOUBLE

| VOID

;

Statements:

Statement

| Statements Statement

;

Statement:

CompoundStatement

| SimpleStatement

| DefinitionStatement

| IfStatement

| WhileStatement

| ForStatement

| DoStatement

| ReturnStatement

;

DefinitionStatement:

VariableDefinition

;

CompoundStatement:

'{' Statements '}'

;

SimpleStatement:

Expression ';'

| ';'

;

IfStatement:

IF '(' Expression ')' Statement

| IF '(' Expression ')' Statement ELSE Statement

;

WhileStatement:

WHILE '(' Expression ')' Statement

;

ForStatement:

FOR '(' ForExpression ';' ForExpression ';' ForExpression ')' Statement

;

DoStatement:

DO Statement WHILE '(' Expression ')' ';'

;

ReturnStatement:

RETURN Expression ';'

| RETURN ';'

;

ForExpression:

Expression

|

;

Expression:

AssignmentExpression

;

AssignmentExpression:

EqualityExpression

| IDENTIFIER '=' EqualityExpression

;

EqualityExpression:

RelationalExpression

| EqualityExpression EQUALITY RelationalExpression

| EqualityExpression INEQUALITY RelationalExpression

;

RelationalExpression:

AdditiveExpression

| RelationalExpression '<' AdditiveExpression

| RelationalExpression '>' AdditiveExpression

| RelationalExpression GREOREQU AdditiveExpression

| RelationalExpression LESSOREQU AdditiveExpression

;

AdditiveExpression:

MultiplicativeExpression

| AdditiveExpression '+' MultiplicativeExpression

| AdditiveExpression '-' MultiplicativeExpression

;

MultiplicativeExpression:

UnaryExpression

| MultiplicativeExpression '\*' UnaryExpression

| MultiplicativeExpression '/' UnaryExpression

| MultiplicativeExpression '%' UnaryExpression

;

UnaryExpression:

PostfixExpression

| '-' UnaryExpression

| '+' UnaryExpression

| PLUSPLUS UnaryExpression

| MINUSMINUS UnaryExpression

;

PostfixExpression:

PrimaryExpression

| IDENTIFIER '(' ArgumentList ')'

| PostfixExpression PLUSPLUS

| PostfixExpression MINUSMINUS

;

PrimaryExpression:

NUMBER\_INT

| NUMBER\_DOUBLE

| IDENTIFIER

| '(' Expression ')'

;

%%

**CSD Dili İçin Soyut Sentaks Ağacının Tasarımı**

CSD Dili için soyut sentaks ağacını parça parça belirleyelim:

1) NUMBER\_INT atomu düğüm aşağıdaki gibi bir yapı ile temsil edilebilir:

typedef struct tagASTNODE\_NUMBER\_INT {

int type;

int number;

} ASTNODE\_NUMBER\_INT;

2) NUMBER\_DOUBLE atomu düğüm aşağıdaki gibi bir yapı ile temsil edilebilir:

typedef struct tagASTNODE\_NUMBER\_DOUBLE {

int type;

double number;

} ASTNODE\_NUMBER\_DOUBLE;

3) IDENTIFER atomu da aşağıdaki gibi bir yapıyla temsil edilebilir:

typedef struct tagASTNODE\_IDENTIFIER {

int type;

char \*id;

} ASTNODE\_IDENTIFIER;

4) Tek operandlı tüm operatörler NODE\_UNARY\_OPERATOR isimli bir yapıyla temsil edilmektedir. Bu yapının içerisinde operatörün türü de vardır:

typedef struct tagASTNODE\_UNARY\_OPERATOR {

int type;

int opType;

ASTNODE \*exp;

} ASTNODE\_UNARY\_OPERATOR;

5) İki operandlı tüm operatörler NODE\_BINARY\_OPERATOR isimli bir yapıyla temsil edilmektedir. Bu yapının içerisinde operatörün türü de vardır:

typedef struct tagASTNODE\_BINARY\_OPERATOR {

int type;

int opType;

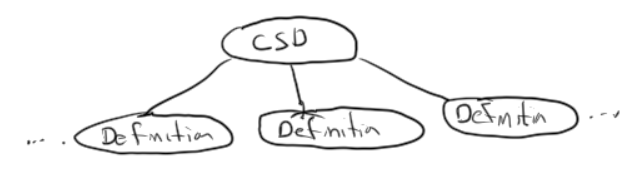
ASTNODE \*leftExp;

ASTNODE \*rightExp;

} ASTNODE\_BINARY\_OPERATOR;

6)

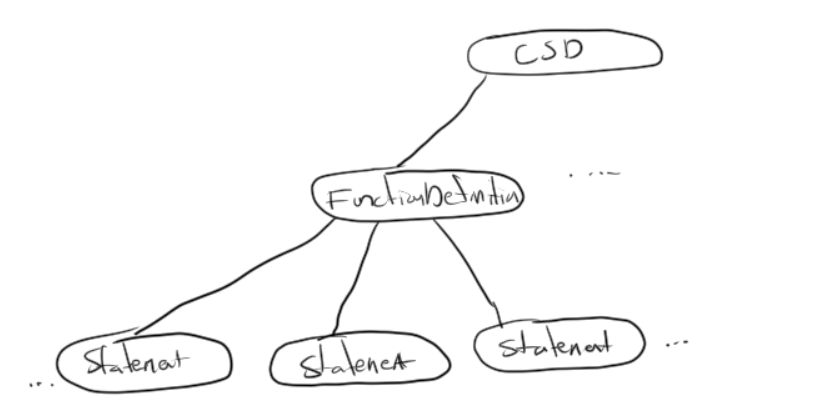
Programımız “Definition”lardan oluşmaktadır. “Definition”lar ikiye ayrılırlar. Biz bu farklı “Defitinion” türünü farklı yapılarla temsil edeceğiz. Ancak sanki aynı türdenmiş gibi bir bağlı listede saklayacağız.



Yukarıdaki şekilde “Definition” aslında “FunctionDefinition” ya da “VariableDefinition” düğümlerini belirtmektedir.

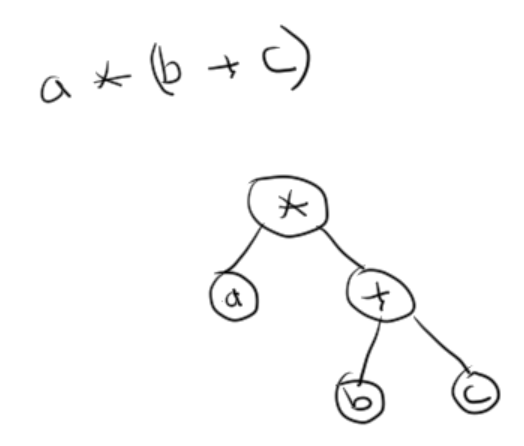
“FunctionDefinition”lar “Statement” lardan oluşmaktadır. Her Statementg için ayrı bir yapı bildirilecektir. Ancak bunlar homojen olarak sanki aynı türdenmiş gibi bir bağlı listede tutulacaktır.

NUMBER atomu için



3) Her “Statement “ kendi özgü ayrı bir veri yapısına sahiptir. Örneğin while döngüsüne ilişkin düğümde while parantezi içerisindeki ifade ve o ifade doğruysa çalıştırılacak deyim listesi bulunur.

4) “Expression”lar da bir ağaç yapısıyla ifade edilebilirler:



Şimdi kabaca ifade ettiğimiz bu soyut sentaks ağacını Bison ile oluşturmaya çalışalım.