D - Programming Language

Yannick Welsch

16. März 2005

Ausarbeitung geschrieben im Rahmen des
Proseminar Programmiersprachen der AG Softwaretechnik
TU Kaiserslautern

Inhaltsverzeichnis

1 Einleitung

1.1 Herkunft

Walter Bright ist Erfinder der Programmiersprache D. Er hat sie seit Dezember 1999 spezifiziert und implementiert. Aus seiner praktischen Erfahrung als Entwickler der Zortech, Symantec und Digital Mars C / C++ Compiler war es ihm möglich den ersten Compiler für D zu schreiben. Dieser Compiler wurde für die Software-Firma Digital Mars[1] entwickelt, er erhielt den Namen dmd (Digital Mars D Compiler).

1.2 Motivation zur Entwicklung der Sprache

Aus seiner Erfahrung als Entwickler von C / C++ Compilern wollte Walter Bright eine Alternative zur Sprache C++ anbieten.
Als grosse Schwäche von C++ sah er die komplexe Spezifikation der Sprache (über 800 Seiten in Buchform) und die schwierige Implementation. Ausserdem bemängelte er die fehlende Basis-Funktionalität in der Sprache (wie z.B. die automatische Speicherverwaltung und Strings), die erst durch Standard- oder Zusatz-Bibliotheken und den Präprozessor ergänzt wurden. Die Fehleranfälligkeit der C++ Programme waren ein weiterer Grund eine neue Sprache zu entwickeln.

1.3 Ziele

> "*D is a practical language for practical programmers who need to get the job done quickly, reliably, and leave behind maintainable, easy to understand code*" – Walter Bright

Als Alternative zu C++ ist das wichtigste Ziel von D den Entwickler produktiver zu machen. D soll dabei folgende Ziele bestmöglichst vereinen:

- Syntax an C,C++,Java angelehnt
- Schnelligkeit und Potenzial von C++
- hohe Skalierbarkeit (von systemnahen Programmen zu komplexen Programmgerüsten)
- leichtgewichtigen und portablen Code
- Unterstützung unterschiedlicher Programmierstile
- Kompatibilität zur C API (Aufruf-Konventionen)
- einfachere und bessere Compiler-Implementationen als für C++
- erhöhte Zuverlässigkeit ermöglicht durch *Design by Contract*[2] und *Unit testing*[3]

D versucht bestehende Konzepte aufzugreifen und eine harmonische Mischung daraus zu erzeugen, wobei dem Programmierer sehr viele syntaktische Freiheiten gelassen werden. Da die Syntax an C,C++,Java angelehnt ist wird sowohl das Erlernen der Sprache D als auch das Portieren von C,C++,Java-Code nach D einfacher.

2 Kurzüberblick über Syntax und Semantik

In den nächsten Unterkapiteln werden herausragende Merkmale der Sprache D vorgestellt und jeweils an einem praxisnahen Beispiel verdeutlicht.

2.1 Organisation

Module sind die Organisationseinheiten in `D`. Ein Modul ist dabei jeweils genau eine `D`-Quelldatei:

```
module beispiel;
```

Module definieren Namensräume und können importiert werden. Dieses `(private) import` geschieht symbolisch im Gegensatz zu `C / C++`. `private import` importiert dabei den Namensraum des importierten Moduls lokal zum Modul in das importiert wird, `import` hingegen ermöglicht das Zusammenfassen von Namensräumen zur späteren "Weitergabe":

```
private import std.stdio;
```

`private` verhindert hier, dass beim Importieren von beispiel in ein neues Modul der ganze std.stdio Namensraum mit importiert wird. Module können ausserdem zu Paketen regruppiert werden, wobei Pakete einfach Verzeichnisse darstellen. Da `D` ohne Präprozessor auskommt, wurden Befehle zur konditionellen (bedingten) Compilierung eingeführt, darunter `version`, `debug`, `static assert` und `deprecated`:

```
version(Windows) {
    void func(){ writef("called:winfunc"); }
}
else version(linux) {
    void func(){ writef("called:linfunc"); }
}
else {
    static assert(0);
}
```

Die Auswahl der Bedingungen erfolgt dann entweder über den Code oder durch Übergabewerte an den Compiler. `version` und `debug` können ausserdem einen Zahlenwert `n` erhalten. Dann wird der Code im `version/debug`-Rumpf eingebunden wenn dem Compiler als Übergabeparameter ein `version/debug`-Wert grösser gleich `n` übergeben wird. `static assert` wird zur Compile-Zeit ausgewertet, wobei der Compiler bei 0 den Compile-Vorgang abbricht.
Um ein ausführbares Programm zu erzeugen, wird eine *main* Methode definiert:

```
int main() {
    debug(4)
        writef("4! ");
    func();
    return 0;
}
```

Der Compiler - Aufruf erfolgt nun z.B. folgendermassen:

```
dmd beispiel -version=Windows -debug=7
```

Dabei wird "`4! called:winfunc`" ausgegeben.
Der Programmierer kann also Software mit unterschiedlicher Anzahl an Features schreiben. Ausserdem muss er sich durch Wegfall des Präprozessors nicht mehr um Mehrfach-imports kümmern (vgl. `C++`). Auch sind bei `D` Vorwärts-Deklarationen überflüssig (Deklarationszeitpunkt = Definitionszeitpunkt!).

2.2 Datentypen / Kompatibilität zu C

Alle Datentypen der C Spezifikation von 1999 wurden in D übernommen[1]. Der D-Compiler erstellt Standard-Objektdateien, kennt auch die C Aufruf-Konventionen, und kann deshalb C Objektdateien einbinden[2]. Um eine Funktion aus einer externen C-Bibliothek zu benutzen muss sie zuerst deklariert weden, kann dann aber gleich benutzt werden:

```
module beispiel2;
extern (C) void sleep(double sleeptime);
int main() {
    sleep(1000);
    return 0;
}
```

Dem Binder wird die externe C-Bibliothek mit übergeben. Mit dem einfachen Einbinden von externem Code vermeidet D das unnötige Neuschreiben von bestehenden C-Bibliotheken. So konnten z.B. mit geringem Aufwand die GTK+ Bibliotheken[9] eingebunden werden.
Zur Erstellung von vollwertigen neuen Typen gibt es in D das Schlüsselwort `typedef`. `alias` hingegen erschafft nur einen Namensersatz für einen Bezeichner. Zur Konvertierung von Typen gibt es neben der impliziten Konvertierung den expliziten `cast`-Befehl. Mit `typedef` definierte neue Typen können zur Überladung von Funktionen benutzt werden:

```
module beispiel3;
private import std.stdio;
typedef int foo;
void myprint(int input) {
    writef("myprint(int) called");
}
void myprint(foo input) {
    writef("myprint(foo) called");
}
int main() {
    foo myvar=cast(foo)4;
    myprint(myvar); //myprint(foo) called
    myprint(6); //myprint(int) called
    return 0;
}
```

Das Erstellen von vollwertigen neuen Typen erhöht die Verständlichkeit des Codes, der Code vermittelt dem Leser mehr als dass es sich hier nur um einen reinen Zahlenwert handelt.

2.3 Datentypen - Felder / Strings

Es gibt 4 verschiedene Arten von Feldern in D:

- Zeiger auf allokierten Speicher	z.B. `int* p;`
- Statische Felder	z.B. `int[3] s;`
- Dynamische Felder	z.B. `int[] d;`
- Assoziative Felder (Hash-Tabellen)	z.B. `int[char[]] x;` oder `int[int] y;`

[1] Dabei haben D-Typen feste Grössen
[2] Da im C-Standard die Grösse der Typen nicht festgelegt ist, muss man hier Vorsicht walten lassen

Strings sind dabei dynamische Felder von char's. ~ ist Konkatenationsoperator. Um zu zeigen wie einfach String-Manipulationen sind, wird nun ein Programm vorgestellt, das die Anzahl von Wörtern, Zeilen und Buchstaben beliebig vieler Dateien ausgibt und ihre Gesamtzahl rechnet. Die `main`-Funktion erhält ein dynamisches Array von Strings (die Dateinamen) als Übergabeparameter:

```
private import std.file, std.stdio;
int main (char[][] args) {
```

Gesamtzahl aller Wörter / Zeilen / Buchstaben:

```
    int w_total=0; int l_total=0; int c_total=0;
```

foreach-Schleifen erlauben einfaches Iterieren über einen Container-Typ. Dabei weist `foreach` dem String *arg* bei jedem Durchlauf ein Element aus *args[1..args.length]* zu. *arg* wird also bei jedem Durchlauf einer der an main übergebenen Dateinamen zugewiesen:

```
    foreach (char[] arg; args[1 .. args.length]) {
```

w_cnt, l_cnt, c_cnt ist die Anzahl aller Wörter / Zeilen / Buchstaben einer Datei. Nun wird der ganze Inhalt der Datei mit Namen *arg* in den String input eingelesen:

```
        int w_cnt=0; int l_cnt=0; int c_cnt=0;
        char[] input;
        input = cast(char[])std.file.read(arg);
```

`foreach` iteriert nun über jeden Buchstaben aus input und bestimmt die Anzahl aller Wörter / Zeilen / Buchstaben in input:

```
        bit inword;
        foreach (char c; input) {
            if (c == '\n')
                ++l_cnt;
            if (c != ' ') {
                if (!inword)
                {
                    inword = true;
                    ++w_cnt;
                }
            } else
                inword = false;
            ++c_cnt;
        }
```

Nun wird die Anzahl aller Wörter / Zeilen / Buchstaben in input ausgegeben und aufaddiert:

```
        writefln(l_cnt~","~w_cnt~","~c_cnt~","~arg);
        l_total += l_cnt; w_total += w_cnt; c_total += c_cnt;
    }
```

Wenn mehrere Dateinamen übergeben wurden, wird die Gesamtsumme ausgegeben:

```
    if (args.length > 2)
        writefln(l_total~","~w_total~","~c_total);
    return 0;
}
```

Dieses Programm wurde sehr einfach in `D` realisiert. String-Manipulationen sind in `D` intuitiv, so können Strings auch in `switch-case` Anweisungen verwendet werden.

2.4 Objektorientierung

Objektorientierung in `D` ist sehr ähnlich zu `Java`. Es gibt nur die Einfachvererbung unterstützt durch Interfaces und abstrakte Klassen. Genau wie in Java gibt es auch eine Klassenhierarchie mit *Object* als Wurzel, wobei Objekte auf dem Heap initialisiert werden (vereinfacht Speicherverwaltung).
Zusätzlich gibt es die Modifizierer[3]:

private: Zugriff auf Modul-Ebene beschränkt

package: erweitert private auf Package-Ebene

protected: id. zu `Java`

public: voller Zugriff innerhalb Ausführungsdatei

export: externer Zugriff möglich

Genau wie in `Java` gibt es noch die Modifizierer `static` und `final`. Ein zusätzliches nützliches Schlüsselwort ist `override`, womit zur Compile-Zeit das korrekte Überladen von Funktionen überprüft wird:

```
class Obst {
    int abc(int x);
    int bar();
}
class Apfel : Obst {
    override
    {
        int abc(int x);
        int bar(char c);
    }
}
```

Hier würde z.B. der Compile-Vorgang abbrechen mit der Fehlermeldung: *error, no bar(char) in Obst*. So werden z.B. nach Veränderungen von Funktionen in einer Oberklasse dem Entwickler der Unterklasse diese Verränderungen beim Compilieren bewusst (Compile-Abbruch mit Fehlermeldung).

2.5 Ressourcenverwaltung

Es gibt 3 Typen der Speicherverwaltung:

- Automatische Speicherverwaltung (Garbage Collector)
- Explizite Speicherverwaltung
  z.B. durch überschreiben von `new` und `delete` mit Hilfe von *malloc* und *free* aus der `C` - Bibliothek
- RAII (Resource Acquisition Is Initialization)
  Dabei wird die Belegung einer Ressource an die Lebensdauer eines Objektes gebunden

Sowohl rein automatische als auch manuelle Speicherverwaltung oder Mischformen beider sind möglich. Um die nebenläufige Programmierung zu erleichtern gibt es ausserdem die Modifier `volatile` (id. zu `C / C++`) und `synchronized` (id. zu `Java`).

[3] Diese Modifizierer sind auch in nicht-objektorientiertem Kontext nutzbar (ausgenommen `protected`)

2.6 Funktionen

Funktionsübergabewerte können durch folgende Parameter gekennzeichnet werden:

"in" : Wert wird ausserhalb der Funktion nicht verändert (default)

"out" : Wert wird innerhalb und ausserhalb der Funktion neu initialisiert und kann ausserhalb verändert werden

"inout" : Wert ausserhalb der Funktion kann gelesen und verändert werden

Diese 3 Varianten ermöglichen verständlicheren, leichter zu optimierenden und weniger fehlerbehafteten Code. So lässt sich z.B. sehr einfach eine *swap* Funktion schreiben die zwei Integer-Werte vertauscht:

```
void swap(inout int val1, inout int val2) {
    int temp= val1;
    val1 = val2;
    val2 = temp;
}
```

Funktionen können auch innerhalb anderer Funktionen definiert werden. Ausserdem gibt es anonyme Funktionen, Funktionen mit beliebig langen Parameterlisten, Funktionszeiger und delegates. Delegates sind dabei Funktionszeiger mit einem zusätzlichen Kontextzeiger. Mit ihnen kann das Observer-Pattern sehr einfach nachgebildet werden (Das Observer-Pattern wurde hier sehr stark vereinfacht). Das Subject beinhaltet die Information `i` und die Funktion *updateall* die es selbst aufruft um den Observer mitzuteilen, dass die Information zur Verfügung steht. Ein `delegate`-Array beinhaltet die Identität der Observer und ihre aufzurufende Funktion bei Verfügbarkeit der Information:

```
class Subject {
    private int i;
    public void delegate(int)[] updatelist;
    private void updateall()
    {
        i=7;
        foreach(void delegate(int) dg; updatelist)
            dg(i);
    }
}
```

Bei Aufruf von *updateall* wird dann die registrierte Funktion eines jeden Observers aufgerufen. Der Observer hat eine (public) Funktion, die bei Verfügbarkeit der Information vom Subject aufgerufen wird.

```
class Observer {
    private int information;
    public void setinformation(int i) { information=i; }
}
```

Im Anwendungs-Beispiel registrieren die Observer sich nun beim Subject indem sie die Addresse ihrer *setinformation* Methode ins `delegate`-Array des Subject hinzufügen. Der zusätzliche Kontextzeiger vom delegate ist in diesem Zusammenhang der `this`-Zeiger auf das jeweilige Observer-Objekt. Das Observer-Objekt meldet sich beim Subject an indem es die Adresse seiner setinformation Methode an das Subject übergibt. Dabei kann jeder Observer eine beliebige Methode übergeben, die Signatur zum delegate-Array muss dabei nur identisch sein. Bei Aufruf von *updateall* werden dann die registrierten Methoden der Observer ausgeführt.
Beispiel:

```
int main()
{
    Subject sub1 = new Subject();
    Observer ob1 = new Observer();
    Observer ob2 = new Observer();
    sub1.updatelist~=&ob1.setinformation;
    sub1.updatelist~=&ob2.setinformation;
    sub1.updateall();
    return 0;
}
```

Dieses Observer-Pattern und delegates finden z.B. Verwendung bei der Erstellung von GUI-Gerüsten (*Message-Handler*).

2.7 Zuverlässigkeit / Sicherheit

`D` implementiert *Design by Contract[2]*[4]. Als Basis-Contract dient die `assert`-Anweisung. Sie wird zur Überprüfung von Klassen-Invarianten und Vor- und Nachbedingungen von Funktionen verwendet. Klassen-Invarianten werden nach jedem Konstruktor-Aufruf und vor und nach jedem (public) Methodenaufruf überprüft (*invariant()* wird aufgerufen). Wenn eine `assert`-Anweisung innerhalb der Invariante zu 0 evaluiert wird eine InvariantException erzeugt. Beispiel:

```
class Tag {
    private int t;
    public this(int t) {
        this.t=t;
    }
    public int get() {
        return t;
    }
    invariant() {
        assert(t>=1 && t<=31);
    }
}
```

Vor- und Nachbedingungen helfen das korrekte Verhalten einer Funktion zu überprüfen, Unit Tests[3] dienen zur Überprüfung des korrekten Verhaltens von (Teil-) Modulen. Bei Aufruf einer Funktion wird der *in*-Teil zuerst überprüft (Vorbedingung), vor Beendigung der Funktion wird dann der *out*-Teil überprüft (Nachbedingung) und bei Fehlverhalten entsprechende Exceptions erzeugt:

```
private import std.math;
long quadratwurzel(long x)
    in {
        assert(x>=0);
    }
    out (resultat) {
        assert(resultat*resultat==x);
    }
    body {
        return std.math.sqrt(x);
    }
```

[4]Contracts sind Bedingungen die an einen Programm-Code gestellt und vom Code eingehalten werden müssen

Unit Tests werden nach Bedarf in die Module eincompiliert und vor Programmstart überprüft.

```
unittest {
    assert(quadratwurzel(4)==2);
}
```

Da *Design by Contract* und *Unit Testing* gleich in die Sprache eingebettet sind, wird es dem Programmierer erleichtert, diese nützlichen Features zu nutzen. Somit werden die Programme zuverlässiger (Fehler werden schneller ausfindig gemacht). Die Fehlerbehandlung ist genau wie in `Java` (mit `try-catch-finally`). Als zusätzliche Sicherheitsfeatures gibt es *Array bounds checking* und *default initializer* Werte. Diese Sicherheitsüberprüfungen können jedoch für Release-Builds per Compiler-Übergabeparameter abgeschaltet werden.

2.8 Templates

[Templates oder Schablonen sind Programmgerüste, die eine vom Datentyp unabhängige Programmierung ermöglichen/[4] [Wikipedia]. Sie dienen der Wiederverwendbarkeit von Code. Wie `C++` unterstützt auch `D` Klassen-Templates und Funktionen-Templates. Template-Parameter können Typen, Werte oder wiederum andere Templates sein[5]. Man kann z.B. eine Template-Funktion `map` erstellen, die über ein Array beliebigen Types iteriert und eine Funktion auf den Elementen aufruft. `foldr` faltet ein Array beliebigen Typs von rechts:

```
module listenmodul;
template Listenfunktionen(T)
{
    void map(void function(inout T) apply, T[] liste) {
        foreach(inout T item; liste)
            apply(item);
    }
    T foldr(T function(T, T) apply, T nel, T[] liste) {
        T rec(T temp, int it) {
            if(it>=0)
                return rec(apply(liste[it], temp), it-1);
            return temp;
        }
        return rec(nel,liste.length-1);
}   }
```

Bei `foldr` sieht man, dass in `D` auch Funktionen innerhalb anderer Funktionen definiert und aufgerufen werden können. Das Template wird nun für den Typ `int` instanziiert und benutzt (alias erschafft einen Namensersatz und verhindert damit wiederholtes Ausschreiben langer Befehle) :

```
private import listenmodul, std.stdio;
int sum(int a, int b) { return a+b; }
int main() {
    int[] i; i~=1; i~=2; i~=3; i~=4;
    alias Listenfunktionen!(int) intlist;
    intlist.map(function void(inout int x) {x++;}, i);
    int resultat = intlist.foldr(&sum, 0, i);
    writef(resultat); //gibt "14" aus
    return 0;
}
```

[5]Im Gegensatz zu `C++` können sogar alle Symbole der Symboltabelle übergeben werden

Der `map`-Funktion wird eine anonyme Funktion übergeben, der `foldr`-Funktionen hingegen die Adresse der Funktion `sum`. Templates können darüber hinaus für verschiedene Spezialisierungen überladen werden (So wird ein von den Übergabe-Typen abhängiger Template-Code instanziiert).

2.9 Weitere Features

- Inline Assembler
  `D` bietet von Haus aus einen Inline-Assembler an, was vorbeugt, dass `D`-Compiler herstellerabhängige Syntax haben, und der Code damit Compiler-portabel ist (feste Prozessorarchitektur vorausgesetzt).

- Mixins
  `D` Mixins helfen um Template-Code für den aktuellen Kontext nutzbar zu machen. Dabei wird der Code eines Template-Rumpfes in den aktuellen Kontext eingefügt und instanziiert. Die letzten Zeilen der Template-Instanziierung hätten also auch folgendermassen geschrieben werden können:

  ```
  mixin Listenfunktionen!(int);
  map(function void(inout int x) {x++;}, i);
  int resultat = foldr(&sum, 0, i);
  ```

  Ausserdem ermöglichen Mixins eine spezielle Form der Aggregation.

- Operatoren überladen
  Fast alle Operatoren können in `D` überladen werden, dabei wird die Kommutativität verschiedener Operatoren berücksichtigt

  ```
  class A { int opAdd(int i); }
  A a;
  a + 1;  // id. zu a.opAdd(1)
  1 + a;  // id. zu a.opAdd(1)
  ```

- vereinfachte Schreibweisen für Strings und grosse Zahlenwerte:

  ```
  file (r"\dm\include\stdio.h");
  int i = 18_446_744_073_709_551_615;
  ```

  anstatt wie in `C++` (aber auch möglich in `D`):

  ```
  file ("\\dm\\include\\stdio.h");
  int i = 18446744073709551615;
  ```

- Array slicing:

  ```
  char[] s1 = "hello world";
  char[] s2 = "goodbye      ";
  s2[8..13] = s1[6..11]; // s2 = "goodbye world"
  ```

- vereinfachte getter/setter Methodenzugriffe

  ```
  class Abc {
      private int myprop;
      void property(int newproperty) { // set
          myprop = newproperty;
      }
      int property() { return myprop; } // get
  }
  ```

kann folgendermassen benutzt werden:

```
Abc a;
a.property = 3;          // id. zu a.property(3)
int x = a.property;      // id. zu int x = a.property()
```

3 Fazit

3.1 Nachteile

- kein dynamisches Nachladen von Klassen (vgl. `Java`)
- zur Zeit kein dynamisches Binden von Programm-Code (vgl. `Java`)
- keine Constraints bezgl. Vererbungshierarchie bei Templates (vgl. `Java 5.0`[5])
- geringe Anzahl in `D` geschriebener Bibliotheken
- mangelnde Literatur in Buchform

3.2 Vorteile

- Syntax schnell und einfach zu erlernen
- vorhandene Debugger können genutzt werden (wegen Standard Objektdateien)
- weniger Komplexität und mehr Basis-Funktionalität in der Sprache (vgl. `C++`)
- Deklarationszeitpunkt = Definitionszeitpunkt
- automatische Speicherverwaltung und Threadunterstützung
- *Design by Contract*[2] / *Unit testing*[3]
- Fehlervermeidung (z.B. `override`)
- *Unicode* Unterstützung für alle Teile des Compilers
- bestehende `C`-Bibliotheken können wiederverwendet werden

Die meisten dieser Vorteile führen dazu dass der Entwickler mit `D` produktiver arbeiten kann als er dies mit `C++` tun würde. Es gibt auch ein `D` *Front End*[6] für die *Gnu Compiler Collection* und somit ist `D` auf den meisten Plattformen nutzbar.

3.3 Persönliche Schlussfolgerung

Das Einsatzgebiet von `D` liegt in der Entwicklung von grossen komplexen Softwaresystemen mit hoher Modularität. Für dieses Gebiet scheint es ein gutes Werkzeug zu sein (statisches Binden ausser Betrachtung). Dies gilt jedoch noch in der Praxis zu zeigen (Es herrscht zur Zeit noch ein grosser Mangel an grossen Softwareprojekten, die in `D` geschrieben werden). Die Compiler für `D` scheinen jedoch ausgereift genug um solche Projekte zu realisieren. Es fehlt jedoch noch an Bibliotheken die in `D` geschrieben wurden. Die wichtigsten Sachen sind jedoch schon in der Standard-Bibliothek oder Zusatz-Bibliotheken[7][8][9] zu finden. Einen guten Start in die `D`-Welt bieten die Seiten von Digital-Mars[1] und Wiki4D[10]. Das *Open Directory*[11] bietet eine Vielzahl an Links zu `D`-Projekten.

4 Literatur

Literatur

[1] offizielle D-Seite von Digital-Mars
http://www.digitalmars.com/d

[2] Design by Contract
http://en.wikipedia.org/wiki/Design_by_contract

[3] Unit testing
http://en.wikipedia.org/wiki/Unit_testing

[4] Templates
http://de.wikipedia.org/wiki/Template_(Programmierung)

[5] Generische Programmierung in Java 5.0
http://de.wikipedia.org/wiki/Generische_Programmierung_in_Java_5.0

[6] GNU D Compiler by David Friedman
http://home.earthlink.net/~dvdfrdmn/d

[7] Dsource.org
http://www.dsource.org

[8] D Template Library
http://www.minddrome.com/produtos/d

[9] DUI - a GTK+ wrapper
http://dui.sourceforge.net

[10] Wiki4D
http://www.prowiki.org/wiki4d/wiki.cgi?FrontPage

[11] Open Directory
http://dmoz.org/Computers/Programming/Languages/D