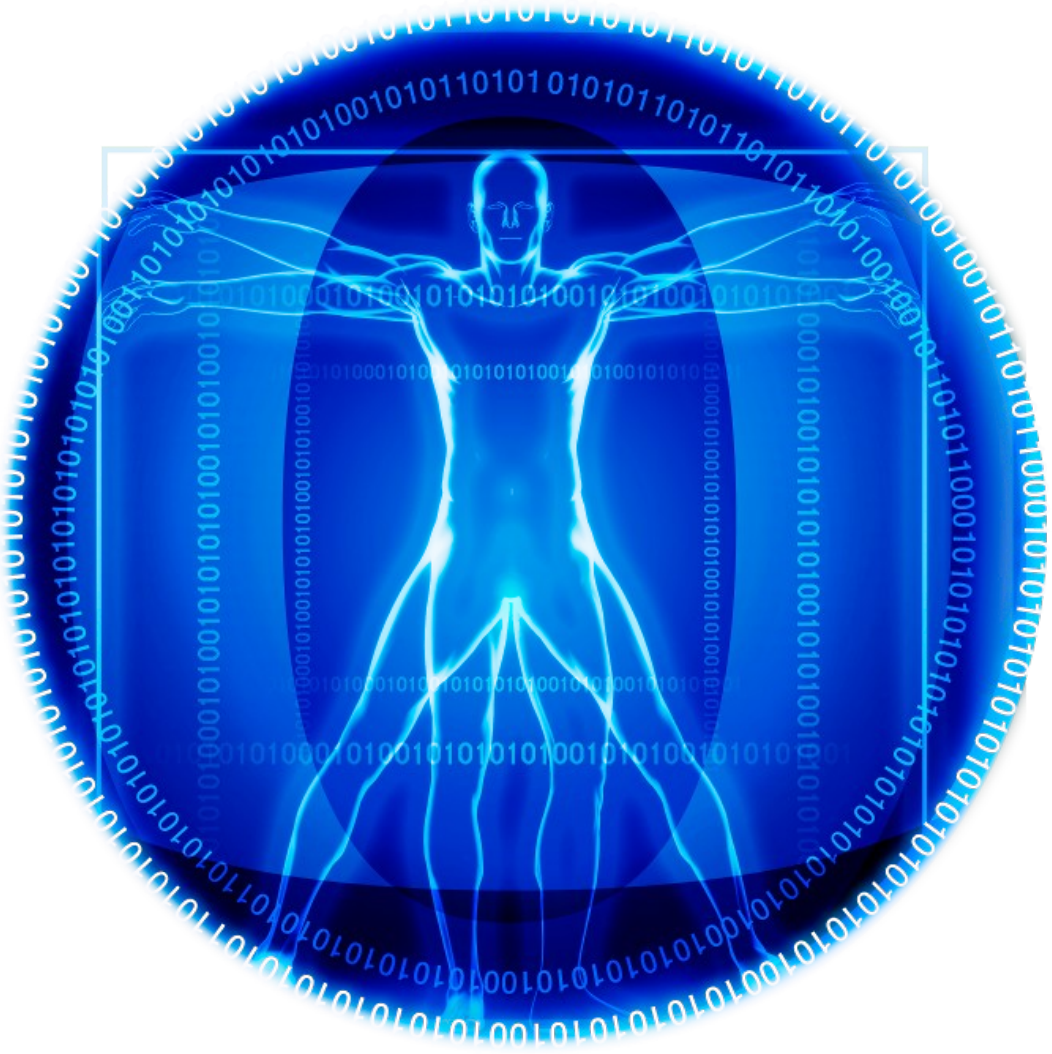
Cybernetics Oriented Programming
(CYBOP)

Handbuch



22.02.2016

Tobias Thurow <tobias.thurow@cs13-2.ba-leipzig.de>

Sebastian Wolff <sebastian.wolff@cs13-1.ba-leipzig.de>

Inhaltsverzeichnis

1 Was ist CYBOP?

CYBOP steht für Cybernetic Oriented Programming und ist eine neue Theorie für die Software-Entwicklung, aufbauend auf Konzepten, die aus der Natur entnommen sind. Es besteht aus den zwei Kernelementen:

- CYBOL
- CYBOI

CYBOL ist eine XML-basierte Anwendungs-Programmiersprache und auch eine Wissens-Spezifikationssprache und somit völlig plattformunabhängig.
CYBOI ist der entsprechende Interpreter, der benötigt wird, um in CYBOL geschriebene Programme auszuführen.

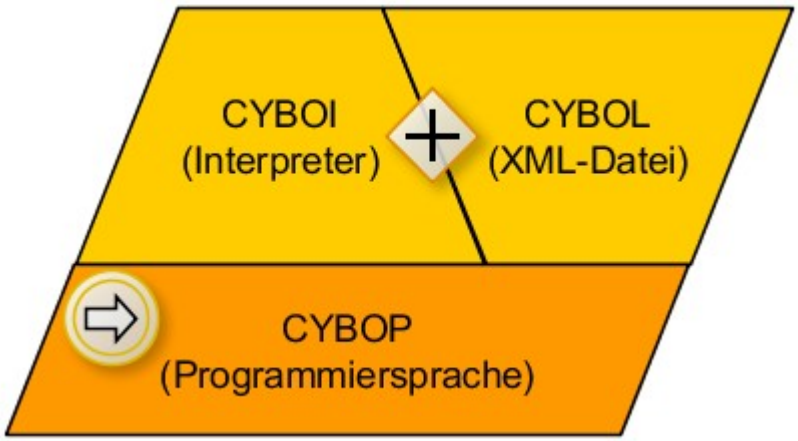


Abbildung 1: CYBOP Komponenten

2 Download

2.1 Voraussetzungen

Um mit CYBOP zu arbeiten benötigt man bestimmte Tools. Dies umfasst folgende:

- autotools
- libtool
- xorg
- xorg-dev
- xlibs-dev

2.2 Download unter Linux/MAC

Unter http://download.savannah.gnu.org/releases/cybop/ stehen verschiedene Versionen zum Download zur Verfügung. Anschließend ist das erhaltene Archiv zu entpacken.

Alternativ kann der neuste Entwicklungsstand über das Savannah Repository heruntergeladen werden. Dafür muss folgender Befehl ausgeführt werden:

```
svn co svn://svn.savannah.nongnu.org/cybop/trunk
```

Es sollte nun folgende Ordnerstruktur vorliegen:

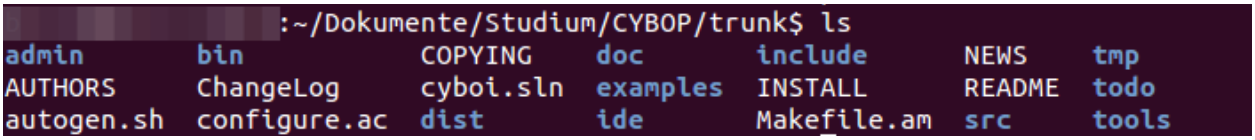


Abbildung 2: Ordnerstruktur

3 Einrichtung

3.1 Build-Prozess

Zur Vereinfachung der nachfolgenden Schritte, sollte der aktuelle Pfad auf das Wurzelverzeichnis der neuen Ordnerstruktur verweisen. Zunächst benötigt die hier liegende `autogen.sh` Ausführungsrechte welche durch `chmod +x autogen.sh` gegeben werden können. Mit dem Befehl `./autogen.sh` kann dieses Skript gestartet werden, was zu einer ähnlichen Ausgabe führt wie:

```
[redacted]:~/Dokumente/Studium/CYBOP/trunk$ chmod +x autogen.sh
[redacted]:~/Dokumente/Studium/CYBOP/trunk$ ./autogen.sh
libtoolize: putting auxiliary files in AC_CONFIG_AUX_DIR, `build-aux'.
libtoolize: copying file `build-aux/ltmain.sh'
libtoolize: Consider adding `AC_CONFIG_MACRO_DIR([m4])' to configure.ac and
libtoolize: rerunning libtoolize, to keep the correct libtool macros in-tree.
libtoolize: Consider adding `-I m4' to ACLOCAL_AMFLAGS in Makefile.am.
configure.ac:49: installing 'build-aux/compile'
configure.ac:23: installing 'build-aux/config.guess'
configure.ac:23: installing 'build-aux/config.sub'
configure.ac:20: installing 'build-aux/install-sh'
configure.ac:20: installing 'build-aux/missing'
src/controller/Makefile.am: installing 'build-aux/depcomp'
```

Abbildung 3: Ausgabe autogen.sh

Nachdem das Skript erfolgreich ausgeführt wurde muss die `configure.sh` ebenfalls gestartet werden. Die Statusmeldungen sollten dabei wie folgt aussehen:

```
[redacted]:~/Dokumente/Studium/CYBOP/trunk$ ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /bin/mkdir -p
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
checking whether make supports nested variables... yes
                        :
checking that generated files are newer than configure... done
configure: creating ./config.status
config.status: creating Makefile
config.status: creating src/Makefile
config.status: creating src/controller/Makefile
config.status: executing depfiles commands
config.status: executing libtool commands
```

Abbildung 4: Ausgabe configure.sh

Der letzte Schritt besteht daraus den make Befehl auszuführen. Bei erfolgreicher Durchführung sieht die Ausgabe folgendermaßen aus:

```
[redacted]:~/Dokumente/Studium/CYBOP/trunk$ make
Making all in src
make[1]: Betrete Verzeichnis '[redacted]/Dokumente/Studium/CYBOP/trunk/src'
Making all in controller
make[2]: Betrete Verzeichnis '[redacted]/Dokumente/Studium/CYBOP/trunk/src/controller'
gcc -DPACKAGE_NAME=\"cybop\" -DPACKAGE_TARNAME=\"cybop\" -DPACKAGE_VERSION=\"0.17.0\" -DPACKAGE_STRING=\"cybop\ 0.17.0\" -DPACKAGE_BUGREPORT=\"christia
n.heller@tuxtax.de\" -DPACKAGE_URL=\"\" -DPACKAGE=\"cybop\" -DVERSION=\"0.17.0\" -DSTDC_HEADERS=1 -DHAVE_SYS_TYPES_H=1 -DHAVE_SYS_STAT_H=1 -DHAVE_STDLI
B_H=1 -DHAVE_STRING_H=1 -DHAVE_MEMORY_H=1 -DHAVE_STRINGS_H=1 -DHAVE_INTTYPES_H=1 -DHAVE_STDINT_H=1 -DHAVE_UNISTD_H=1 -DHAVE_DLFCN_H=1 -DLT_OBJDIR=\".li
bs/\" -DHAVE_LIBM=1 -DHAVE_LIBGL=1 -DHAVE_LIBGLU=1 -DHAVE_LIBX11=1 -DHAVE_LIBPTHREAD=1 -DHAVE_LIBXCB=1 -DSTDC_HEADERS=1 -DHAVE_SYS_WAIT_H=1 -DHAVE_ARPA
_INET_H=1 -DHAVE_FCNTL_H=1 -DHAVE_LOCALE_H=1 -DHAVE_NETINET_IN_H=1 -DHAVE_STDDEF_H=1 -DHAVE_STDLIB_H=1 -DHAVE_STRING_H=1 -DHAVE_SYS_SOCKET_H=1 -DHAVE_T
ERMIOS_H=1 -DHAVE_UNISTD_H=1 -DHAVE_WCHAR_H=1 -DHAVE__BOOL=1 -DHAVE_STDBOOL_H=1 -DHAVE_UNISTD_H=1 -DHAVE_CHOWN=1 -DHAVE_STDLIB_H=1 -DHAVE_MALLOC=1 -DHA
VE_STDLIB_H=1 -DHAVE_REALLOC=1 -DHAVE_MEMSET=1 -DHAVE_SETLOCALE=1 -DHAVE_SOCKET=1 -DHAVE_STRTOL=1 -I.    -I/usr/include -DGNU_LINUX_OPERATING_SYSTEM -g
 -O2 -MT cyboi.o -MD -MP -MF .deps/cyboi.Tpo -c -o cyboi.o cyboi.c
mv -f .deps/cyboi.Tpo .deps/cyboi.Po
/bin/bash ../../libtool  --tag=CC   --mode=link gcc -I/usr/include -DGNU_LINUX_OPERATING_SYSTEM -g -O2   -o cyboi cyboi.o  -lxcb -lpthread -lX11 -lGLU
-lGL -lm
libtool: link: gcc -I/usr/include -DGNU_LINUX_OPERATING_SYSTEM -g -O2 -o cyboi cyboi.o  -lxcb -lpthread -lX11 -lGLU -lGL -lm
make[2]: Verlasse Verzeichnis '[redacted]/Dokumente/Studium/CYBOP/trunk/src/controller'
make[2]: Betrete Verzeichnis '[redacted]/Dokumente/Studium/CYBOP/trunk/src'
make[2]: Für das Ziel »all-am« ist nichts zu tun.
make[2]: Verlasse Verzeichnis '[redacted]/Dokumente/Studium/CYBOP/trunk/src'
make[1]: Verlasse Verzeichnis '[redacted]/Dokumente/Studium/CYBOP/trunk/src'
make[1]: Betrete Verzeichnis '[redacted]/Dokumente/Studium/CYBOP/trunk'
make[1]: Für das Ziel »all-am« ist nichts zu tun.
make[1]: Verlasse Verzeichnis '[redacted]/Dokumente/Studium/CYBOP/trunk'
```

Abbildung 5: Ausgabe make

3.2 Funktionstest

Wenn der Buildprozess erfolgreich abgeschlossen wurde, ist im Verzeichnis `src/controller` der CYBOP Interpreter (CYBOI) zu finden.

Im Verzeichnis examples sind Beispielanwendungen vorhanden. Um eines dieser Programme auszuführen muss zunächst in diesen Ordner gewechselt werden. Von hier aus können die Beispielanwendungen folgendermaßen gestartet werden:

```
../src/controller/cyboi projektname/run.cybol
```

Eine Beispielanwendung ist hierbei das „Hello-World“-Programm, welches wie folgt aufgerufen wird:

```
../src/controller/cyboi helloworld/run.cybol
```

Dies sollte zu folgender Ausgabe führen:

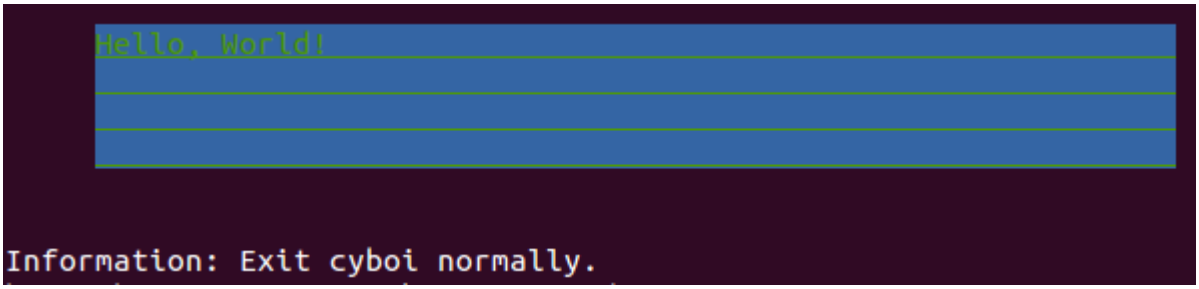


Abbildung 6: Ausgabe "Hello, World!"

Alternativ lässt sich auch ein Symlink anlegen um den Interpreter auch ohne konkrete Referenzierung global erreichen zu können.

```
[redacted]:~/Dokumente/Studium/CYBOP/trunk$ ln src/controller/cyboi /usr/bin/cyboi
```

Abbildung 7: CYBOI als Symlink

4 Aufbau eines CYBOP-Programms

CYBOP ist eine XML-basierte Programmiersprache die aus verschiedenen Knoten besteht. Jedes Programm besitzt einen Wurzelknoten, dem beliebig viele Kindknoten untergeordnet sind. Diese Knoten besitzen die Bezeichnung `<node>`. Der Kern der Sprache besteht in der Verwendung der Attribute. Aus Attributen und Knoten werden so Modelle geformt, mit denen der Programmierer arbeiten kann.

Als Beispiel wird hier zunächst das „Hello-World“-Programm erstellt.

```
<node name="startup" channel="inline" format="maintain/startup" model="">
   <node name="channel" channel="inline" format="meta/channel" model="terminal"/>
</node>
```

Code 1: Terminal erstellen

Zuerst muss das Terminalfenster initialisiert werden. Dazu wird dem Wurzelknoten ein neuer Knoten hinzugefügt:

Jeder Knoten besitzt die Attribute Name, Channel, Format und Model. Der Name dient hier lediglich zur Beschreibung des Knoten. Einige Namen spezifizieren die Eigenschaften des Mutterknoten jedoch näher. Der Channel beschreibt die Ressource, aus der die Information des Knotens ausgelesen wird. Inline beschreibt hierbei das Auslesen aus dem Speicher.

Das Format definiert den Datentyp oder eine Funktion des Knotens. `Maintain/Startup` beschreibt die Erstellung eines bestimmten Objektes. Dieses Objekt wird hier im Kindknoten genauer spezifiziert.

Über das Model-Attribut kann man, entsprechend dem Format, weitere Werte definieren. In dem Knoten „startup“ ist keine Angabe zum Model nötig.

Im Kindknoten wird hier per Name `channel` der Ausgabekanal gesetzt. Durch das Model `terminal` wird eine Konsole initialisiert.

Das fertige Programm sollte folgendermaßen aussehen:

```
<node>
   <node name="startup" channel="inline" format="maintain/startup" model="">
      <node name="channel" channel="inline" format="meta/channel" model="terminal"/>
   </node>
   <node name="print" channel="inline" format="communicate/send" model="">
      <node name="channel" channel="inline" format="meta/channel" model="terminal"/>
      <node name="encoding" channel="inline" format="meta/encoding" model="utf-8"/>
      <node name="language" channel="inline" format="meta/language"
            model="message/cli"/>
      <node name="format" channel="inline" format="meta/format" model="text/plain"/>
      <node name="message" channel="inline" format="text/plain" model="Hello
            World!"/>
      <node name="newline" channel="inline" format="logicvalue/boolean"
            model="true"/>
   </node>
   <node name="shutdown" channel="inline" format="live/exit" model=""/>
</node>
```

Code 2: Hello World - Anwendung

Durch das Format `communicate/send` wird ein Datenstrom erstellt. Über diesen soll anschließend der Text „Hello World!“ ausgegeben werden. Dafür muss dieser Befehl weiter spezifiziert werden. Hierfür werden die in `name` eingetragenen Eigenschaften verändert:

- channel – Ausgabekanal
- encoding – Zeichenkodierung der Ausgabe
- language – Sprache der Ausgabe
- format – Datentyp der Ausgabe
- message – Ausgabe
- newline – am Ende der Ausgabe einen Zeilenumbruch einfügen

Als Ausgabekanal wird im Beispiel das Terminal genutzt, wobei die Zeichenkodierung UTF-8 genutzt wird. Als Language wird `message/cli` (Command Line Interface) genutzt, über das einfacher Text - `text/plain` - ausgegeben werden soll. Durch die `message` „Hello World!“ wird die Ausgabe konfiguriert und der `newline` Parameter fügt am Ende einen Zeilenumbruch hinzu.

Dies führt zur Ausgabe der Zeile „Hello World!“ auf einer Konsole. Am Schluss muss die Anwendung beendet werden, was mit Hilfe des im Knoten „shutdown“ verwendeten Formats `live/exit` realisiert wird.

5 Variablen

5.1 Erstellen und Initialisieren von Variablen

Um eine Variable zu erstellen, muss zunächst Speicherplatz reserviert werden. Dies geschieht mit dem Format `memorise/create`.

```
<node name="create_value" channel="inline" format="memorise/create" model="">
   <node name="name" channel="inline" format="text/plain" model="value"/>
   <node name="format" channel="inline" format="meta/format" model="number/integer"/>
   <node name="element" channel="inline" format="text/plain" model="part"/>
</node>
```

Code 3: Erstellen einer Variable

Der Name bestimmt hierbei, unter welchem Namen auf die Variable zugegriffen werden kann. Mittels Format wird der Typ der Variable angegeben. Element bestimmt zuletzt, dass die Variable in das Speichermodell eingefügt wird.

Anschließend muss der Variable ein Wert zugewiesen werden. Dies ist mittels `communicate/receive` möglich.

```
<node name="initialise_value" channel="inline" format="communicate/receive" model="">
   <node name="channel" channel="inline" format="meta/channel" model="inline"/>
   <node name="encoding" channel="inline" format="meta/encoding" model="utf-8"/>
   <node name="language" channel="inline" format="meta/language" model="text/cybol"/>
   <node name="format" channel="inline" format="meta/format" model="number/integer"/>
   <node name="sender" channel="inline" format="text/plain" model="1"/>
   <node name="message" channel="inline" format="path/knowledge" model=".value"/>
</node>
```

Code 4: Initialisieren einer Variable

Der Channel bestimmt hierbei, woher die Eingabe bezogen wird. Neben dem Model `inline` ist es hier auch möglich `file` zu wählen, um aus einer externen Datei zu lesen. Das Encoding sollte standardmäßig auf `utf-8` gesetzt werden. Als Language wird hier `text/cybol` genutzt, das Format ist `number/integer`, wodurch eine ganze Zahl als Eingabe gelesen wird. Der Sender hat hierbei, unabhängig des Formates, immer den Typ `text/plain`. Im Model werden die Daten, ggf. ein Array aus mehreren Werten, eingegeben. Zuletzt beschreibt `message` den Empfänger der Nachricht, in diesem Fall die Variable `value`, welche durch das Format `path/knowledge` als solche erkannt wird.

5.2 Zugriff auf Variablen

Variablen werden in einer Baumstruktur erstellt. Diese ist davon abhängig, wie die Variablen erstellt werden. In diesem Beispiel wurde die Variable direkt an den Wurzelknoten angefügt. Im Kapitel Domainreferenzierung finden sich weitere Informationen, wie diese Baumstruktur genutzt werden kann.

```
<node name="print" channel="inline" format="communicate/send" model="">
   <node name="channel" channel="inline" format="meta/channel" model="terminal"/>
   <node name="encoding" channel="inline" format="meta/encoding" model="utf-8"/>
   <node name="language" channel="inline" format="meta/language"
   model="message/cli"/>
   <node name="format" channel="inline" format="meta/format" model="number/integer"/>
   <node name="message" channel="inline" format="path/knowledge" model=".value"/>
   <node name="newline" channel="inline" format="logicvalue/boolean" model="true"/>
</node>
```

Code 5: Zugriff auf Variable

Um auf Variablen zugreifen zu können müssen diese zunächst erstellt und initialisiert werden. Das Model beinhaltet den Variablennamen während das Format angibt, von welchem Datentyp die Variable ist. Mit dem Format `path/knowledge` wird auf den Datenspeicher referenziert. Um beispielsweise auf die zuvor erstellte Variable zu verweisen, wird als Model `.value` verwendet.

5.3 Datentypen

Es gibt verschiedene Datentypen die verwendet werden können. Diese sind verschiedenen Knoten zugeordnet wodurch man unterschiedlich auf diese verweisen muss. Datentypen wie `double`, `integer` und `byte` sind unter `number/` verfügbar. Da logische Aussagen hier keiner eindeutigen Zahl entsprechen ist der Datentyp `boolean` unter `logicvalue/` zu erreichen. Auch Zeichenketten kann man durch `text/plain` erstellen. Alle verfügbaren Datentypen sind der API zu entnehmen.

6 Operationen

6.1 Arithmetik

Rechenoperationen sind dem Knoten `calculate` untergeordnet. Im folgendem Beispiel werden zwei Variablen miteinander addiert. Hierbei werden die zwei Operanden voneinander unterschieden. Der Knoten `result` enthält die Variable auf welcher der zweite Operand addiert werden soll. Dieser wird dann im Knoten `operand` angegeben. Der `type` gibt zum Schluss Informationen darüber welcher Datentyp für die Addition verwendet werden soll.

```
<node name="add" channel="inline" format="calculate/add" model="">
   <node name="result" channel="inline" format="path/knowledge" model=".result"/>
   <node name="operand" channel="inline" format="number/integer" model="5"/>
   <node name="type" channel="inline" format="meta/type" model="number/integer"/>
</node>
```

Code 6: Addition

In diesem Beispiel wird die Variable `result` um 5 erhöht. Wenn man bei dem Knoten `operand` nun das `format` auf `path/knowledge` und das `model` auf `.result` ändern würde, dann würde man die Variable mit sich selbst addieren. Alle Rechenoperationen sind nur für Zahlen verwendbar.

6.2 Arithmetische Funktionen

Unter `calculate` sind folgende Rechenoperationen verfügbar:

- `add`
- `substract`
- `divide`
- `multiply`
- `negate`
- `absolute`

Auch hier werden die Knoten `result`, `operand` und `type` verwendet.

6.3 Stringoperationen

6.3.1 Stringverkettung

Operationen zur Stringmanipulation sind unter dem Knoten `modify` verfügbar. Um Beispielsweise zwei Strings miteinander zu verketten wird `append` verwendet. Wie schon bei den Rechenoperationen gibt es hier drei Knoten. `Destination` übernimmt hierbei die Rolle von `result` und `source` die Rolle des Operanden. Dem String, der in `destination` angegeben wird, wird der String aus `source` angefügt. Wenn die Variable `.value` mit „app“ initialisiert wird dann enthält diese nach der Ausführung des folgendem Beispiels den String „append“.

```
<node name="append" channel="inline" format="modify/append" model="">
   <node name="destination" channel="inline" format="path/knowledge" model=".value"/>
   <node name="source" channel="inline" format="text/plain" model="end"/>
   <node name="type" channel="inline" format="meta/type" model="text/plain"/>
</node>
```

Code 7: Stringverkettung

6.3.2 Ausschneiden von Teilstrings

Da man zwei Strings miteinander verketten kann gibt es auch die Möglichkeit Strings auszuschneiden. Diese Funktion ist über `modify/overwrite` erreichbar. Zu den bereits bekannten Knoten kommen nun zwei weitere. Der erste neue Knoten ist der `source_index`. Dieser gibt den Anfangsindex des auszuschneidenden Strings an. Der zweite Knoten, `count`, gibt an wie viele Zeichen ab dem Anfangsindex ausgeschnitten werden sollen. Im folgendem Beispiel wird aus der Zeichenfolge „TicTacToe“, das „Tac“ herausgeschnitten und in der Variable `.value` gespeichert.

```
<node name="cut" channel="inline" format="modify/overwrite" model="">
   <node name="destination" channel="inline" format="path/knowledge" model=".value"/>
   <node name="source" channel="inline" format="text/plain" model="TicTacToe"/>
   <node name="type" channel="inline" format="meta/type" model="text/plain"/>
   <node name="source_index" channel="inline" format="number/integer" model="3"/>
   <node name="count" channel="inline" format="number/integer" model="3"/>
</node>
```

Code 8: Ausschneiden von Teilstrings

6.4 Vergleiche

Um Variablen miteinander zu vergleichen benötigt man den Knoten `compare`. Hierunter befinden sich folgende Vergleichsoperationen :

- `equal`
- `unequal`
- `greater`
- `smaller`
- `greater-or-equal`
- `smaller-or-equal`

Während `equal` und `unequal` für Text und Zahlen verwendbar ist, sind die anderen Operatoren nur für Zahlen verwendbar. Die Knoten die für alle `compare` Funktionen verwendet werden sind: `left` und `right` für die jeweiligen Vergleichselemente, `type` für den Datentyp, `result` für das Ergebnis und `selection` für den Bereich der verglichen werden soll.

Bei folgendem Beispiel wird die Variable `.value` mit dem String „Text“ verglichen. Das Ergebnis wird in die Variable `.bool` geschrieben.

```
<node name="compare" channel="inline" format="compare/equal" model="">
   <node name="left" channel="inline" format="path/knowledge" model=".value"/>
   <node name="right" channel="inline" format="text/plain" model="Text"/>
   <node name="type" channel="inline" format="meta/type" model="text/plain"/>
   <node name="result" channel="inline" format="path/knowledge" model=".bool"/>
   <node name="selection" channel="inline" format="text/plain" model="all"/>
</node>
```

Code 9: Vergleich

7 Referenzen

Um in Cybol Funktionen verwenden zu können, muss man sich mit dem Referenzmodell vertraut machen. Primitive Datentypen können direkt initialisiert werden, während komplexe Datentypen in externen Dateien ausgelagert werden müssen.

Die in diesem Handbuch verwendeten Datei- und Domainreferenzen unterscheiden sich von den Referenzierungen, die im Ordner `examples/manual` zu finden sind. Die Referenzierungen müssen immer relativ zum Aufrufsort des CYBOL Interpreters stattfinden. Bei den Beispielen die hier erklärt werden geht man davon aus, das der Aufruf des Interpreters innerhalb des jeweiligen Ordners stattfindet, wobei im Gegensatz zu den Beispielen unter `examples/manual,` der Interpreter vom `examples` Verzeichnis aus aufgerufen wird.

7.1 Dateireferenzierung

In diesem Beispiel werden zwei Dateien verwendet. Die `fileref.cybol` startet das Programm, ruft eine referenzierte Datei auf und beendet danach das Programm. In der `print.cybol` befindet sich nur die Ausgabe eines Textes auf der Konsole.

Mit `format="flow/sequence"` ist es möglich eine Funktion aufzurufen. Im Kindknoten `model` wird dafür der `channel="file"` verwendet. Das Format `element/part` gibt an, dass die ausgelesene Datei einen komplexen Datentyp beinhaltet. Das `model` dieses Knotens gibt den Pfad zur referenzierten Datei an. Dies führt dazu, dass die gesamte Referenzdatei ausgeführt wird.

```
<node>
   <node name="startup" channel="inline" format="maintain/startup" model="">
      <node name="channel" channel="inline" format="meta/channel" model="terminal"/>
   </node>
   <node name="call_reference" channel="inline" format="flow/sequence" model="">
      <node name="model" channel="file" format="element/part" model="print.cybol"/>
   </node>
   <node name="shutdown" channel="inline" format="live/exit" model=""/>
</node>
```

Code 10: fileref.cybol

```
<node>
   <node name="print" channel="inline" format="communicate/send" model="">
      <node name="channel" channel="inline" format="meta/channel" model="terminal"/>
      <node name="encoding" channel="inline" format="meta/encoding" model="utf-8"/>
      <node name="language" channel="inline" format="meta/language"
      model="message/cli"/>
      <node name="format" channel="inline" format="meta/format" model="text/plain"/>
      <node name="message" channel="inline" format="text/plain" model="Called by
      Reference"/>
      <node name="newline" channel="inline" format="logicvalue/boolean"
      model="true"/>
   </node>
</node>
```

Code 11: print.cybol

7.2 Domainreferenzierung

Das folgende Beispiel verwendet wieder die `print.cybol` als Funktion.

Die Domainreferenzierung ermöglicht es eine Hierarchiestruktur zu erstellen. Hierbei werden Knoten erzeugt, welche anderen Knoten untergeordnet werden können. Jeder Knoten kann dabei beliebig verwendet werden um unterschiedliche Teil-Ganzes-Beziehungen darzustellen.

Man kann Beispielsweise folgende Knoten erstellen um das Prinzip der objektorientierten Programmierung zu übernehmen:

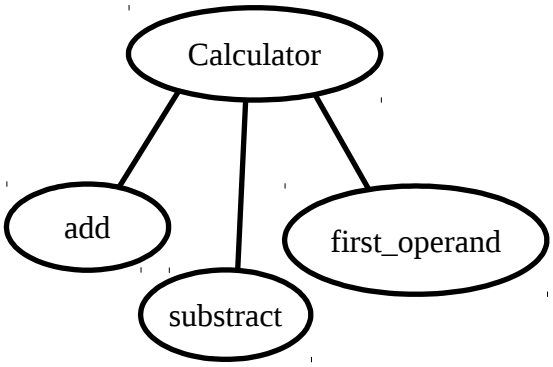


Abbildung 9: Domainbeispiel 1

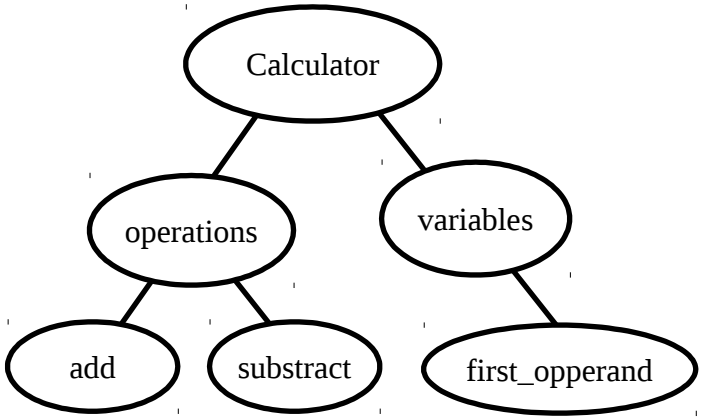


Abbildung 8: Domainbeispiel 2

Die linke Abbildung kommt dem Modell der objektorientierten Programmierung nahe. Hier gibt es einen Wurzelknoten dem Variablen und Funktionen untergeordnet sind, so wie es Klassen gibt die Methoden und Variablen enthalten.

In der rechten Abbildung ist sichtbar, was mit der Knotenstruktur, die CYBOP bietet, möglich ist. Man kann Teile im Gesamtkonstrukt genauer einordnen. Im Beispiel wurde hier zwischen Operation und Variable unterschieden. Die Knotenstruktur ist beliebig erweiterbar sodass man sehr viel Freiraum hat eine geeignete Struktur für sein Programm zu finden.

In dem nächsten Abschnitt wird folgendes Modell erstellt:

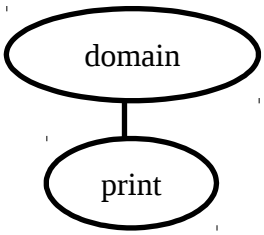


Abbildung 10: Domain domain.print

Um dies zu erreichen, muss zunächst mittels `memorise/create` ein Objekt namens `domain` erzeugt werden. Damit es möglich ist, einen anderen Knoten an diesen anzufügen, ist es notwendig, diesen vom Typ `element/part` zu erzeugen. Es ist jedoch nicht notwendig, das Domainobjekt zu initialisieren.

Als nächstes soll die Funktion `print` unter den Domain-Knoten angehangen werden. Dabei ist zu beachten, dass das Format `element/part` sein muss, da dies einem komplexen Datentyp entspricht. Mittels des Knotens `whole` kann angegeben werden, an welchen Knoten das neue Objekt angehangen werden soll. In diesem Beispiel wird als Pfad `.domain` aus dem Speichermodell gewählt, welches vorher bereits angelegt wurde.

Als nächstes muss die Funktion `print` initialisiert werden. Da die Methode aus einem extra File ausgelesen werden muss, ist es zunächst notwendig, den `Channel` auf `File` zu setzen. Um das File mittels CYBOL zu interpretieren muss weiterhin `language` auf `text/cybol` gesetzt werden. Das Format ist entsprechend des erstellten Objektes vom Typ `element/part`. Als Sender muss hier der Pfad zu der entsprechenden Code-Datei angegeben werden, in welcher die Print-Funktion zu finden ist. Dafür wurde hier erneut die Funktion aus dem vorherigen Beispiel genutzt. Zuletzt wird als `message` der Pfad im Speicherbaum angegeben, der dem zuvor erstellten Objekt für die Funktion entspricht.

Nach Erstellung dieser Funktion ist es möglich, sie aus dem Speichermodell aufzurufen. Dafür wird im Knoten `call_reference` ein Aufruf ausgeführt. Im Gegensatz zum Aufruf des Files aus dem vorherigen Beispiel, wird hier auf `.domain.print` verwiesen. Dadurch wird der Code aus der Print-Funktion ausgeführt.

Bei öfterem Ausführen einer Funktion bietet sich hier der Vorteil, dass es nicht notwendig ist, jedes mal erneut die Datei einzulesen.

```
<node>
   <!-- node: domain -->
   <node name="domain" channel="inline" format="memorise/create" model="">
      <node name="name" channel="inline" format="text/plain" model="domain"/>
      <node name="format" channel="inline" format="meta/format"
      model="element/part"/>
      <node name="element" channel="inline" format="text/plain" model="part"/>
   </node>

   <!-- node: print -->
   <node name="function_print" channel="inline" format="memorise/create" model="">
      <node name="name" channel="inline" format="text/plain" model="print"/>
      <node name="format" channel="inline" format="meta/format"
      model="element/part"/>
      <node name="element" channel="inline" format="text/plain" model="part"/>
      <node name="whole" channel="inline" format="path/knowledge" model=".domain"/>
   </node>

   <node name="init_print" channel="inline" format="communicate/receive" model="">
      <node name="channel" channel="inline" format="meta/channel" model="file"/>
      <node name="encoding" channel="inline" format="meta/encoding" model="utf-8"/>
      <node name="language" channel="inline" format="meta/language"
      model="text/cybol"/>
      <node name="format" channel="inline" format="meta/format"
      model="element/part"/>
      <node name="sender" channel="inline" format="text/plain" model="print.cybol"/>
      <node name="message" channel="inline" format="path/knowledge"
      model=".domain.print"/>
   </node>
   <node name="startup" channel="inline" format="maintain/startup" model="">
      <node name="channel" channel="inline" format="meta/channel" model="terminal"/>
   </node>
   <node name="call_reference" channel="inline" format="flow/sequence" model="">
      <node name="model" channel="inline" format="path/knowledge"
      model=".domain.print"/>
   </node>
   <node name="shutdown" channel="inline" format="live/exit" model=""/>
</node>
```

Code 12: Domainreferenzierung

8 Kontrollstrukturen

8.1 If-Anweisungen

Um in CYBOL eine If-Anweisung zu erstellen, ist es notwendig, das ausschlaggebende Kriterium für das Einschlagen eines bestimmten Zweiges als boolean Variable zu speichern. Die auszuführenden Aktionen müssen entweder als File oder als Funktion im Speicher vorhanden sein.

Zunächst muss ein Knoten mit dem Format `flow/branch` erstellt werden. Dieser erhält den Unterknoten `criterion`, in welchem auf die boolean Variable verwiesen wird.

Anschließend können ein `true` und ein `false` Knoten erstellt werden, denen jeweils eine Funktion zugeordnet wird. Dafür können die 2 Möglichkeiten der Referenzierung genutzt werden.

```
<node name="compare" channel="inline" format="flow/branch" model="">
   <node name="criterion" channel="inline" format="path/knowledge" model=".flag"/>
   <node name="true" channel="file" format="element/part" model="true.cybol"/>
   <node name="false" channel="file" format="element/part" model="false.cybol"/>
</node>
```

Code 13: If-Else Anweisung

Im folgenden Beispiel werden abhängig vom Flag die `true.cybol` oder die `false.cybol` ausgeführt.

8.2 Schleifen

Um eine Schleife zu erstellen, ist es nötig, eine Boolean zu erzeugen, welche als Abbruchbedingung dient. Diese muss über den Speicherbaum der Anwendung erreichbar sein. Weiterhin wird eine Funktion benötigt, die bei jedem Schleifendurchlauf ausgeführt werden soll. Hier muss darauf geachtet werden, dass die Abbruchbedingung in dieser Funktion gesetzt wird, da ansonsten eine Endlosschleife entsteht.

Im folgenden Beispiel wurde als Wurzelknoten `.loop` erstellt. Anschließend wurden unter diesen die Integer `count` und die Boolean `break` gehangen. Der Aufruf der Schleife geschieht mit dem Format `flow/loop` und den Unterknoten `break` und `model`. Im `break` Knoten wird dabei auf die boolean Variable verwiesen, `model` ruft eine Funktion per Dateireferenz auf.

```
<node name="print_numbers" channel="inline" format="flow/loop" model="">
   <node name="break" channel="inline" format="path/knowledge" model=".loop.break"/>
   <node name="model" channel="file" format="element/part" model="loop.cybol"/>
</node>
```

Code 14: Schleife

Anschließend wird die `loop.cybol` aufgerufen, bis die Variable `.loop.break` auf true gesetzt wird. Im Beispielprogramm wird hier zunächst „Counter: [count]“ ausgegeben. Anschließend wird die Variable `.loop.count` mittels der Additionsfunktion inkrementiert.

Anschließend wird hier geprüft, ob der Zähler größer als 10 ist. Falls dies zutrifft, wird die Abbruchbedingung `.loop.break` auf true gesetzt, was zum Beenden der Schleife führt.

```
<node name="increment_loop_count" channel="inline" format="calculate/add" model="">
   <node name="result" channel="inline" format="path/knowledge" model=".loop.count"/>
   <node name="operand" channel="inline" format="number/integer" model="1"/>
   <node name="type" channel="inline" format="meta/type" model="number/integer"/>
</node>
<node name="compare_loop_count" channel="inline" format="compare/greater" model="">
   <node name="result" channel="inline" format="path/knowledge" model=".loop.break"/>
   <node name="left" channel="inline" format="path/knowledge" model=".loop.count"/>
   <node name="right" channel="inline" format="number/integer" model="10"/>
   <node name="type" channel="inline" format="meta/type" model="number/integer"/>
   <node name="selection" channel="inline" format="text/plain" model="all"/>
</node>
```

Code 15: Schleifen-Abbruch-Bedingung

Abbildungsverzeichnis

Codeverzeichnis