

YAB BASIC per Haiku

revisione 0.1 del 07/02/2012



Interpreter

Guida (non in vendita) realizzata da Haiku Italia www.haiku-os.it
Articoli originali in lingua inglese realizzati da **Jan Bungeroth a.k.a. Jan_64**
Traduzione di Giuseppe Gargaro



Jan Bungeroth a.k.a. Jan_64 PhD-Student at the Chair of Computer Science 6 at RWTH Aachen - University of Technology, Germany

Link

- <http://www.team-maui.org/projects/yab/>
- http://yab-interpretor.sourceforge.net/index_en.htm
- <http://www.yabasic.de/>

Yab è sviluppato dal Team Maui , un gruppo di utenti Haiku con base in Germania.

Potete scaricare l'interprete yab da www.haikuware.com

Indice

[Introduzione a YAB](#)

[Yab guida per principianti N°1](#)

[Yab guida per principianti N°2](#)

[yab-C++ Tutorial \(Tutorial per aggiungere nuovi comandi a yab\)](#)

Introduzione a YAB (articolo comparso su BeOSNews)

Il sistema operativo Haiku è scritto principalmente in C++ e C e così la gran parte delle sue applicazioni. Per la gran parte delle attività il C++ è un ottimo linguaggio potente e leggibile. Comunque per applicazioni più piccole c'è talvolta il desiderio di un linguaggio più semplice, in questo caso YAB risulta un'ottima soluzione. Il BASIC (**B**eginners **A**ll-purpose **S**ymbolic **I**nstruction **C**ode), è un linguaggio disegnato inizialmente per aiutare le persone a comprendere le basi della programmazione su un computer, ma risulta adeguato anche alla realizzazione di applicazioni. Oggigiorno ci sono moltissime varianti BASIC, come pureBasic o il conosciutissimo Visual Basic della Microsoft. L'interprete basic YAB di jan__64 è una variante BASIC specializzata per BeOS/Zeta/Haiku ed è ampiamente basata su yabasic, una libera implementazione del BASIC per Windows e Linux, rispetto a cui è arricchito da alcuni elementi delle API per la GUI di BeOS/Haiku.

Man mano che yab progredisce e diventa un linguaggio completo sarà possibile sviluppare ogni tipo di applicazione. Attualmente l'interprete è quasi finito e anche l'IDE per YAB (scritto nello stesso linguaggio) è a buon punto, è un interprete davvero interessante e l'IDE è fatto davvero bene rendendo molto facile sviluppare applicazioni, soprattutto gestionali o software che non richiedono grandi prestazioni.. il software viene avviato con degli script che lanciano l'interprete facendole sembrare delle applicazioni compilate. Lo sviluppo di YAB è portato avanti dal TEam Maui.

La struttura di base di yab è come già detto derivata da Yabasic, ciò significa che molti programmi sviluppati per Windows, Linux (Unix) e per la Playstation 2 possono essere facilmente convertiti in BeOS e Zeta; uno dei vantaggi dello sviluppo in Yab è la licenza GPL e artistica di questo interprete i suoi diretti concorrenti DarkBasic(Windows), Blitz Plus (Windows), Pure Basic (Windows, Linux, Amiga) costano attorno ai 50 euro; un'altro vantaggio dello sviluppo in yab è la possibilità di scrivere un'interfaccia grafica grazie allo sviluppo in yab degli elementi della GUI di BeOS/Haiku.. naturalmente le prestazioni non sono paragonabili a quelle del C++.

Un esempio:

Per mostrare quanto sia semplice realizzare piccole interfacce con yab, eccovi il codice della più basilare di tutte le applicazioni. Le persone che hanno esperienza con la programmazione in Visual Basic riconosceranno lo stile del codice, ad esempio per l'assenza del carattere ";". Nel complesso il codice sembra più quello di un linguaggio di scripting che quello di linguaggi quali Java, C o C++.

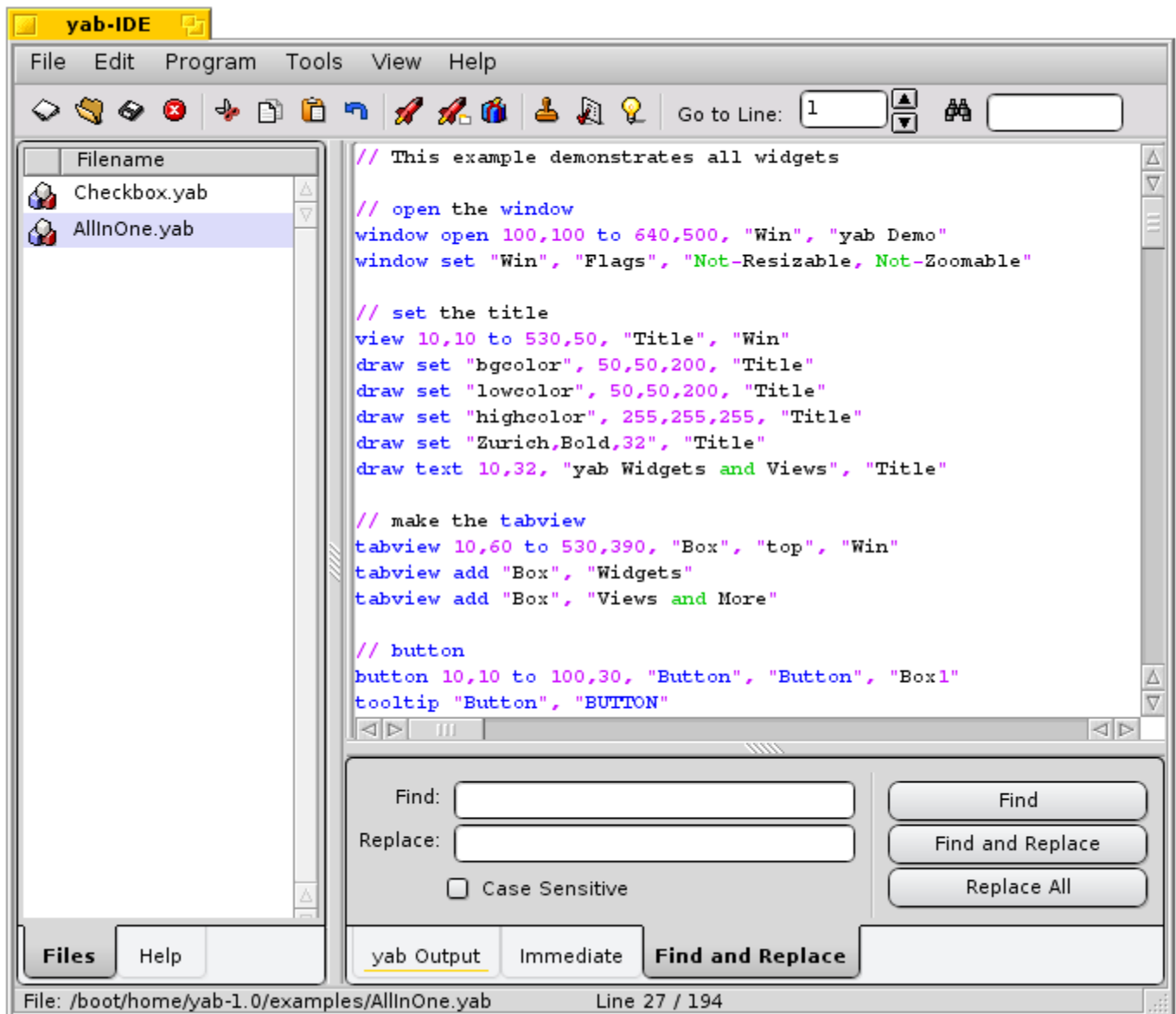
```
HelloWorld in yab:
  window open 200,200 to 500, 600, "MainView", "Hello World"
  draw text 100,100, "Hello World", "MainView"
  sleep 5
window close "MainView"
```

Le applicazioni Yab consistono di file sorgente con comandi come quelli presenti nel codice qui sopra che vengono avviati grazie all'interprete. Attualmente l'interprete è un file di circa 1 MB, e perciò può essere facilmente distribuito insieme all'applicazione. Tipicamente dopo lo sviluppo compilereste i vostri sorgenti in un binario per distribuirlo. Con yab è possibile creare ogni tipo di controllo di un'interfaccia Haiku presente nelle API, più altri controlli come Date/Time picker, ed un check button. Al di sotto dei controlli di un'interfaccia ci sono tutte le altre cose che servono ad un'applicazione loop, condizioni, funzioni aritmetiche ecc. Chiunque ha scritto codice con un altro tipo di BASIC sarà a suo agio con la struttura del codice di yab. Uno dei punti di forza del codice BASIC è che è molto facile, semplicemente leggendolo, capire cosa accade dal punto di vista della programmazione. Perciò yab ha una curva di apprendimento con molti meno passaggi di quella necessaria ad apprendere il C++, ed ognuno può creare applicazioni

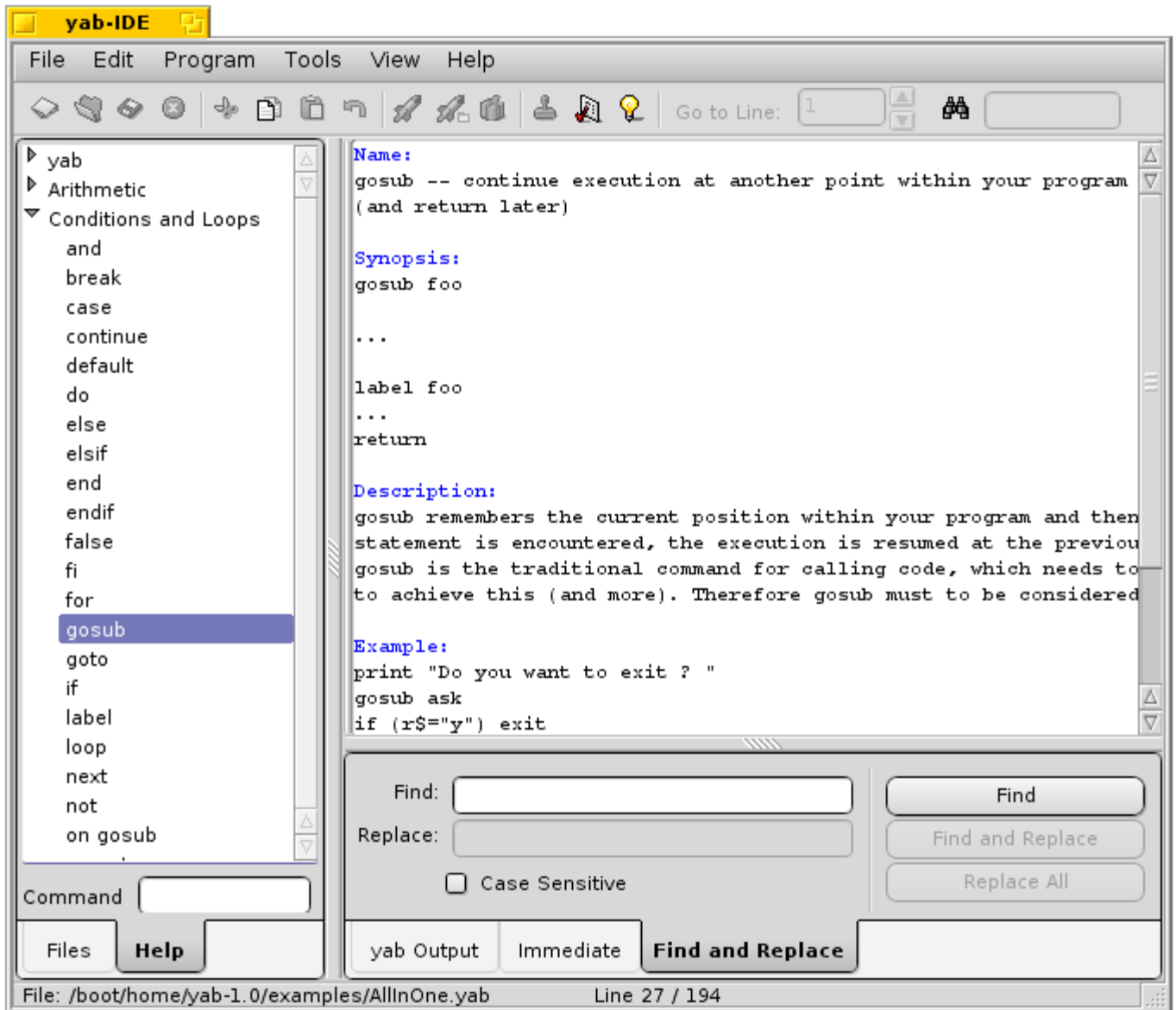
nell'arco di minuti. Un linguaggio come yab permetterà di attrarre studenti ed hobbisti per la realizzazione di applicazioni per Haiku.

L' IDE

Sviluppare applicazioni per yab consiste essenzialmente nello scrivere codice e testarlo nell'interprete, tutto quello che vi occorre è un editor di testo. Ma yab è fornito anche di un IDE che può avviare istantaneamente le vostre applicazioni per testarle, creare binari del vostro codice ed ha un editor che supporta la specifica sintassi di yab evidenziando e completando il codice automaticamente. Qui sotto potete vedere uno screenshot dell' IDE. Sulla sinistra sono elencati i file attualmente aperti, sulla destra c'è l'editor del codice.



Oltre a un bel editor di codice, yab-IDE ha anche un buon sistema di documentazione, ci sono descrizioni per tutti i comandi e le funzioni disponibili che potete utilizzare nel vostro codice.



Con yab-IDE potete avviare i vostri progetti con un click, per testarli. Ciò significa che non dovrete più aspettare che il compilatore finisca quando state testando la vostra applicazione. Potete anche avviare la vostra applicazione in una shell, o tenere traccia del debug output nella Output view. Inoltre quello che l'IDE può fare per voi è inserire le vostre applicazioni in un binario che può essere avviato sotto BeOS/ Haiku. Ciò rende invisibile il sorgente alle persone ed inoltre non dovranno necessariamente installare l'interprete per avviare le vostre applicazioni. L'IDE contiene un color picker ed è tutte le funzioni per la ricerca e sostituzione, auto indentazione e riformattazione del sorgente.

Flyab

A seguito dell'introduzione delle specifiche API di BeOS in Yab la compatibilità con gli altri sistemi operativi è stata rotta. Comunque esiste un nuovo progetto "Flyab" per permettere alle applicazioni yab di funzionare su diverse piattaforme. Con Flyab sarà possibile avviare le applicazioni yab su Windows, Mac, Linux e naturalmente Haiku. Per ottenere questo risultato Flyab usa il toolkit 'FL' (Fast Light), e allo stato attuale sono stati implementati un 40% dei comandi originali di yab.

[- torna all'indice -](#)

YAB guida per principianti N°1

Creare la finestra di una applicazione

Creare la finestra di una applicazione è semplice, dovete fornire solo l'istruzione *window open* e aggiungere alcuni dati riguardo dimensione, nome e indicazioni interne.

```
window open 50,50 to 600,500, "View", "WindowName"
```

Viene aperta una finestra, che inizia a 50 pixel dal bordo sinistro dello schermo e a 50 pixel dal bordo superiore. Dopo *to* vengono indicati il bordo destro e quello inferiore della finestra. Così otteniamo una finestra di dimensioni 550 pixel in larghezza e 450 pixel in altezza. Come potete vedere non viene data la dimensione della finestra, ma solo le sue coordinate sullo schermo.

Dopo i dati relativi alla dimensione della finestra segue il nome (qui: "View") per uso interno (*internal*). Con questo nome potete richiamare la finestra per esempio per chiuderla o inserire qualcosa in entrata.

WindowName indica il nome, che verrà visualizzato nella linguetta per contraddistinguere la finestra. Cioè il nome, che vedrà l'utente.

Inloop: Durevole richiesta di funzioni

Di seguito creiamo *un Inloop*, che ci è d'aiuto nell'esecuzione richieste ricorrenti nel programma. Tutto quello che è situato in questo loop (ciclo) del programma viene eseguito continuamente come richiesto. Tutto ciò che sta fuori dal loop si avvia solo una volta o quando viene richiamato. Qui va notato che tutto quello che è prima del loop viene eseguito prima dell'avvio del loop e tutto quello che è contenuto nel loop solo quando richiamato.

```
inloop = true
while(inloop)
  msg$ = message$
  switch msg$
```

```
    Here the entries are inserted into the loop.
```

```
  end switch
wend
```

Chiudere la finestra dalla TAB è un buon esempio, perché questa funzione dovrebbe essere selezionabile in qualunque momento nella sequenza del programma. Inserite il seguente codice nella parte indicata nel loop con " Here the entries are inserted into the loop.". Utilizziamo l'istruzione *case*, che inizia con l'indicazione del nome e termina con *break*.

```
case "View:_QuitRequested|"
  window close "View"

break
```

Quando viene cliccato *close* sulla TAB, con il nome della finestra (*View*) e viene passata una domanda pertinente (qui *_QuitRequested*).

Tutto ciò che si trova tra *case "View:_QuitRequested|"* e *break*, verrà eseguito una volta chiamata la sequenza. Poiché vorremmo chiudere la finestra, con *window close "View"* la finestra *View* viene chiusa.

Barra del menù nella finestra

Nella maggior parte dei casi la finestra ha una barra dei menù. Per avere questa barra dei menù, dovete aggiungere il seguente codice sopra *inloop* e dopo l'apertura della finestra del programma, affinché sia sempre visibile.

```
menu "Program", "Quit", "Q", "View"
```

Inoltre con questo esempio ci occupiamo della terminazione del programma. Con questo codice creiamo nel menù la voce *Program* e dopo questo punto si trova il punto secondario *Quit* (termina). L'indicazione della *Q* è l'indicazione della scorciatoia di tastiera (Alt + Q) con cui uno può terminare il programma senza utilizzare il menù. L'ultima istruzione *View* è la finestra in cui va collocato il menù. Per arrivare alla formazione di altre voci del menù, devono essere inserite ulteriori linee, che sono costruite con lo stesso metodo illustrato. Cambiando il primo nome viene automaticamente aggiunta una seconda voce al menù. Lo stesso vale per i sotto menù.

Per l'opzione menù va aggiunta un'ulteriore entrata in *Inloop*. Che appare come segue:

```
case "View:Program:Quit|"
    window close "View"
break
```

Poiché chiamiamo una chiusura della finestra sopra la barra dei menu, l'edizione del comando segue la sequenza della selezione di menu scelta. La finestra del programma (*View*) su cui si trova il menù, la voce del menù *Program* e il sotto menù *Quit*.

Creare View aggiuntive

Per non chiudere sempre la finestra del programma per caricare una nuova vista, è raccomandabile creare una ulteriore *View* alla stessa view principale. Ciò dovrebbe cominciare approssimativamente... 10 pixel sotto (nel caso fuori) una barra del menù nella finestra del programma il margine superiore.

```
view 0,10 to 550,450, "View1", "View"
```

Creare una *View* è simile ad aprire una finestra. Noi forniamo una *View* con il nome *View1* sulla finestra del programma *View*. La *View* comincia in base al formato della finestra di programma sul lato sinistro da 0 pixel e dal margine superiore con 10 pixel (a causa della barra dei menu). Il bordo destro e quello inferiore sono indicati dopo *to*.

La *View1* è inserita dove dovrebbe essere indicata, in questo caso è direttamente sotto la finestra del programma.

Mostrare un testo

Ora mostreremo del testo su questa *View1*.

```
draw text 20, 20, "this is a text edition", "View1"
```

Con *draw text* indichiamo che vorremmo aggiungere del testo. Con le specifiche *20.20* indichiamo le coordinate del testo sulla *View*. Cioè 20 pixel dal margine sinistro e 20 pixel dal margine superiore. Il testo

che segue è il testo, che è indicato sulla View. Qui non viene fornita nessuna automatic line-makeup, così ciò va fornito, adattando il testo per indicare la finestra (View). Alla fine segue l'indicazione delle Views, su cui va indicato il testo.

Mostrare un grafico

Per mostrare grafici dobbiamo aggiungere il codice seguente sotto le Views, su cui gradiremmo questi annunci (View1).

```
err = draw image 150,30, "/Path/to/Graphic/graphic.jpg", "View1"
```

Con *err = draw image* indichiamo che vorremmo inserire un grafico. Come nel caso del testo, seguono le coordinate, a cui va indicato lo schema. Secondo l'indicazione della posizione dell'immagine, vengono indicati il percorso del file e il file (*Path/to/Graphic/graphic.jpg*), che vorreste mostrare. Inoltre alla conclusione di questa istruzione segue l'indicazione delle Views, su cui lo schema deve essere indicato.

Creare un bottone

Che cosa è un programma senza superfici di selezione per le specifiche funzioni del programma. Sia per aprire ulteriori finestre di programma o per effettuare varie mansioni, un tasto è utilizzato molto spesso.

```
button 400, 400 to 450, 420, "internal_Button name", "shown_Button name", "View1"
```

Noi forniamo un *Button* con 400 pixel del Viewborder di sinistra e 400 pixel del Viewborder superiore. Il formato dei tasti deriva dai dati seguenti (450 pixels) sul bordo destro e (420 pixels) su quello inferiore del bottone. come mostra l'esempio il nome del bottone per uso interno è seguito dal nome che viene mostrato sul bottone.

La funzione dei tasti è indicata come per la chiusura del programma, in *Inloop* come una istruzione *case* .

```
case "View1:internal_Button name|"
    window close "View"
break
```

Qui come prima con gli altri *cases* viene indicata la View e il nome interno del bottone. Inoltre in questo caso usiamo l'istruzione *window close* per terminare il programma. Il tasto può anche mostrare per esempio il testo supplementare, il tasto, grafica o aprire e/o chiudere una finestra o una vista ulteriore.

Test del programma nel terminale

Se volete testare il vostro programma potete avviare il codice sorgente direttamente in una finestra del terminale.

```
yab /Path/to/source code.yab
```

Basta inserire *yab* in una finestra del terminale seguito da uno spazio. Poi trascinate (drag and drop) il file

sorgente sul terminale così che il percorso venga mostrato, premete Enter per eseguire il programma.

Nel caso che state usando yab IDE, potete avviare il programma dalla MenuBar selezionate *Program* e poi con *Run in Terminal* lanciate il programma.

Legare il codice del programma

Il vostro programma è pronto, il prossimo passo è legarlo (*bind*). Legandolo impedirete ad altre persone di visualizzare il codice del programma così solo voi potrete guardare e modificare il codice sorgente.

Aprirete una finestra del terminale e inserite quanto segue:

```
yab -bind "Name of the program" /Path/to/source code.yab
```

Con *yab -bind* dichiarate che volete legare il codice sorgente. Il nome del programma è il nome del file che trovate nella vostra cartella home. Il codice sorgente può essere aggiunto come prima trascinandolo (drag and drop).

Help durante la programmazione

Vista la somiglianza con YABASIC, è sempre raccomandato cercare le soluzioni nella loro [Documentazione](#).

yab viene fornito con del codice esempio, che mostra differenti funzioni, potete trovarlo nella vostra cartella yab.

Su BeSly c'è un [yab ProgramHelp](#) (in tedesco), dove sono descritte molte funzioni.

[- torna all'indice -](#)

YAB guida per principianti N°2

Creiamo un'applicazione

Durante questo tutorial creeremo un'applicazione per realizzare immagini bfs.

Il primo passo è pensare: di quali funzioni ha bisogno questa applicazione? Un BFS Creator ha bisogno di un'opzione per impostare il nome e la dimensione del file immagine. Abbiamo bisogno di conoscere il percorso (path) alla cartella scelta per scrivere l'immagine.

Il secondo passo è riflettere sulla costruzione dell'applicazione. Fatelo con gli occhi degli utenti di questa applicazione.

- L'avvio del programma
- Interfaccia per impostare nome e dimensione del file immagine
- Un pulsante per avviare la creazione dell'immagine
- Apertura dell'albero dei file per selezionare la destinazione del file immagine
- Creare l'immagine
- Informazione per l'utente, che l'immagine è stata creata

Pianificazione dell'applicazione:

```
#!/boot/home/config/bin/yab

screenWidth = peek("desktopwidth")
screenHeight = peek("desktopheight")

dim part$(1)

inloop = true
while(inloop)
msg$ = message$
if (split(msg$, part$(1), ":|") > 2) then
    PartTwo$ = part$(2)
    PartThree$ = part$(3)
fi
if (msg$ <> "") print msg$

switch msg$
    default:
end switch

if(window count<1) inloop = false

wend
```

Questo template codice sorgente è più grande di quanto avete bisogno.Vi ho scritto più informazioni, affinché possiate usarlo per i vostri prossimi progetti.

All'inizio otteniamo le dimensioni dello schermo con `screenWidth = peek("desktopwidth")` e `screenHeight = peek("desktopheight")`. Prima la larghezza dello schermo (desktopwidth) e poi l'altezza (desktopheight). Utilizziamo queste informazioni per posizionare la main window nel mezzo dello schermo.

Poi creiamo un Array con *dim part\$(1)* con il valore 1. Fatto questo possiamo iniziare con l'inloop. Voi lo conoscete già dalla "YAB guida per principianti N°1". Con *msg\$ = message\$* copiamo l'informazione *message\$* nella variabile *msg\$*.

```
if (split(msg$, part$(1), ":|") > 2) then
    PartTwo$ = part$(2)
    PartThree$ = part$(3)
fi
if (msg$ <> "") print msg$
```

Poi dividiamo la variabile *msg\$*. Stiamo dividendo da ":" al segno "|". Il segno ":" separa le parti del messaggio. Il segno "|" è la fine del messaggio.

Con queste variabili si ottengono le informazioni per la funzione che l'utente sta chiamando.

Ad esempio: Avete un pulsante chiamato Button. Quando utilizzate il terminale per testare il codice sorgente ottenete la seguente informazione: "Button|". Tale informazione può essere facilmente utilizzata con un *case* per avviare una funzione.

La finestra del programma

Creiamo una finestra ed utilizziamo le informazioni sulla risoluzione dello schermo per posizionarla in mezzo allo schermo:

```
window open ((screenWidth/2)-170), ((screenHeight/2)-80) to ((screenWidth/2)+170),
((screenHeight/2)+80), "createbfs", "BFS Image Creator"
    window set "createbfs", "MinimumTo", 340,160
    window set "createbfs", "MaximumTo", 340,160
```

Questo crea una finestra con una larghezza (*width*) di 340 pixel ed una altezza (*height*) di 160 pixel. Per ottenere il punto di mezzo della Finestra, prendiamo i valori *ScreenHeight* e *ScreenWidth* e li dividiamo per due. Poi prendiamo la metà della dimensione della finestra voluta e lo sottraiamo al valore diviso per ottenere la posizione sinistra della finestra. Facciamo lo stesso per il lato destro, ma non sottraiamo il valore, ma aggiungiamo il valore. Così abbiamo, tra il valore aggiunto e sottratto, la larghezza voluta della finestra. Facciamo lo stesso con la *ScreenHeight*, aggiungiamo e sottraiamo la metà della finestra voluta alla *ScreenHeight* (left side= middle of the screen -170 | middle of the screen +170 = right side).

Impostiamo la massima e minima grandezza della finestra utilizzando *window set*. Così l'utente potrà impostare la finestra tra questi due valori. Se impostate entrambi i valori alla stessa dimensione l'utente non potrà ridimensionare la finestra.

Interfaccia

Il prossimo passo è creare un'interfaccia per l'applicazione.

```
draw text 20, 40, "Size of the Imagefile:", "createbfs"
textcontrol 150,25 to 270,60, "bfssize", "", "", "createbfs"
draw text 275, 40, "MB", "createbfs"
```

```
draw text 20, 65, "Name of the Imagefile:", "createbfs"
textcontrol 150,50 to 270,65, "bfssize", "", "", "createbfs"
draw text 275, 65, ".image", "createbfs"
```

```
button 120,110 to 220,130, "savebfs", "create", "createbfs"
```

All'inizio di questo tutorial stavamo pensando alle informazioni che l'utente deve inserire per il file immagine. Queste erano la dimensione dell'immagine, il suo nome e il posto dove deve essere creato. Creiamo due *textcontrols*, uno per la dimensione ed un'altro per il nome del file immagine. Il comando

per il `textcontrol` è come quello di un pulsante (button). Impostate `lefthand site` e `top margin site` del `textcontrol` e passateli al `righthand and lower edge site` del `textcontrol`. La prossima informazione è l' `id` del `textcontrol`. E' il nome che l'applicazione utilizza al suo interno. Poi impostate il nome mostrato del `textcontrol` (qui non inseriamo un nome perché per fare questo in questo tutorial, utilizziamo `draw text`). L'ultima informazione è la view su cui collochiamo il `textcontrol`.



Non usiamo il nome del `textcontrol` possiamo impostarlo nel comando `textcontrol`, perché questo nome è direttamente davanti al `textcontrol`, così non avremmo un'interfaccia con un buon aspetto poi il `textcontrol` e il suo nome non hanno lo stesso allungamento.

Davanti e dietro il `textcontrol` utilizziamo un `draw text` così che l'utente sappia che deve inserire il testo qui. Scriviamo `.MB` dietro il primo `textcontrol` così l'utente sa che deve inserire il valore in megabytes. Dietro il secondo scriviamo l'estensione per il file immagine (`.image`) così l'utente sa che l'applicazione usa questa informazione come nome dell'immagine. Ok in BeOS, Haiku e ZETA non siamo tenuti ad impostare un'estensione ma avere un'estensione consente di vedere più rapidamente di che tipo di file si tratta.

L'ultimo oggetto dell'interfaccia è il pulsante per attivare il processo di creazione dell'immagine.

Avviare le funzioni del programma

Così siamo arrivati al punto che dobbiamo creare l'immagine BFS. A tal proposito useremo lo shell tool `dd` per creare un file immagine raw e `mkbfs` per inizializzarlo con il filesystem bfs.

Noi non possiamo fare questo senza le informazioni che l'utente ha inserito nei `textcontrols`

Perciò inseriamo il seguente codice sotto lo `switch msg$` :

```
case "savebfs|"
    bfssize$=textcontrol get$ "bfssize"
    bfsname$=textcontrol get$ "bfsname"
    bfsimage$ = FILEPANEL "Save-File", "Save", "/boot/home", bfsname$+".image"
    Output$=system$("/boot/beos/bin/dd if=/dev/zero of="+bfsimage$+" bs=1024k
count="+bfssize$)
    Output$=system$("mkbfs 2048 "+bfsimage$+"; sync")
    ALERT "Imagefire are created!", "Ok", "info"
break
```

Conoscete la funzione `case` dalla "YAB guida per principianti N°1".

Creiamo una variabile `bfssize$` ed otteniamo con `textcontrol get$ "bfssize"` la dimensione dell'immagine. Otteniamo informazioni da `textcontrol bfssize` utilizzando `textcontrol get$` e salvandole in `bfssize$`.

Facciamo lo stesso per il nome del file immagine. Creiamo anche una variabile ma la chiamiamo `bfsname$`. Otteniamo informazioni da `textcontrol bfsname` con `textcontrol get$` e le salviamo in `bfsname$`.

Dopo ciò, abbiamo le informazioni che sono state inserite dall'utente.

Il prossimo passaggio è ottenere la cartella in cui va inserito il file immagine. Per questo utilizziamo il comando `FILEPANEL`. A tale scopo creiamo una variabile che chiamiamo `bfsimage$`. Con `filepanel` e le informazioni ad esso necessarie apriamo una finestra per navigare il sistema. La chiamiamo `Save-File` per gli usi interni e `Save` come nome da mostrare. Dietro questo c'è il percorso (`path`) alla cartella da cui la finestra di navigazione deve partire (qui `/boot/home`). Qui l'utente entra il nome per l'immagine che viene

salvato nella variabile. La stringa completa della variabile è il percorso(path) con il nome del file immagine e l'estensione.

Con `Output$=system$("/boot/beos/bin/dd if=/dev/zero of="+bfsimage$+" bs=1024k count="+bfssize$)` (con `system$` potete utilizzare le applicazioni fuori da `yab`.) creiamo un file immagine raw `bfsimage$` di dimensione `bfssize$`.

Inizializziamo `Output$=system$("/boot/beos/bin/dd if=/dev/zero of="+bfsimage$+" bs=1024k count="+bfssize$)` il file immagine raw `bfsimage$` con il `filesystem bfs`.

Alla fine del processo di creazione dell'immagine utilizziamo il comando `ALERT` per informare gli utenti che il file immagine è stato creato. Il primo testo è l'informazione per l'utente "`Imagefile is created!`". Il secondo è il nome del pulsante nella finestra alert e l'ultimo è il tipo di informazione (in questo caso una `info`).

Cosa manca?

Aggiungo alcune cose dal primo tutorial "YAB guida per principianti N°1". Includo l'opzione `quit` per la finestra, inserisco un `menu` con informazioni sullo sviluppatore, una opzione `quit option` o altro.

Legare (Bind) il codice del programma

Se il vostro programma è pronto, il prossimo passo è effettuare il binding del codice per fare in modo che altre persone non possano leggerlo, così soltanto voi potrete cambiare e guardare il codice.

Aprire una finestra del terminale ed inserite quanto segue:

```
yab -bind "Name of the App" /Path/to/Sourcecode.yab
```

Con `yab -bind` dichiarate che volete effettuare il binding del codice sorgente. Il nome del programma è il nome del file che potete trovare nella cartella home. Il codice sorgente può essere aperto come prima con il drag and drop.

Questo è uno screenshot del bind source:



Il codice completo:

```
#!/boot/home/config/bin/yab

screenWidth = peek("desktopwidth")
screenHeight = peek("desktopheight")

window open ((screenWidth/2)-170), ((screenHeight/2)-80) to ((screenWidth/2)+170),
((screenHeight/2)+80), "createbfs", "BFS Image Creator"
    window set "createbfs", "MinimumTo", 340,160
    window set "createbfs", "MaximumTo", 340,160

draw text 20, 40, "Size of the Imagefile:", "createbfs"
textcontrol 150,25 to 270,60, "bfssize", "", "", "createbfs"
draw text 275, 40, "MB", "createbfs"

draw text 20, 65, "Name of the Imagefile:", "createbfs"
textcontrol 150,50 to 270,65, "bfsname", "", "", "createbfs"
draw text 275, 65, ".image", "createbfs"

button 120,110 to 220,130, "savebfs", "create", "createbfs"

dim part$(1)

inloop = true
while(inloop)
    msg$ = message$

    if (split(msg$, part$(1), ":|") > 2) then
        PartTwo$ = part$(2)
        PartThree$ = part$(3)
    fi

    if (msg$ <> "") print msg$

    switch msg$

    case "savebfs|"
        &nbsp;bfssize$=textcontrol get$ "bfssize"
        bfsname$=textcontrol get$ "bfsname"
        bfsimage$ = FILEPANEL "Save-File", "Save", "/boot/home", bfsname$+".image"

        Output$=system$("/boot/beos/bin/dd if=/dev/zero of="+bfsimage$+" bs=1024k
count="+bfssize$)
        Output$=system$("mkbfs 2048 "+bfsimage$+"; sync")

        ALERT "Imagefile are created!", "Ok", "info"
        break

    default:
end switch

if(window count<1) inloop = false

wend
```

yab-C++ Tutorial (Tutorial per aggiungere nuovi comandi a yab)

Introduzione

Questo tutorial presuppone conoscenze di base su yab, C, C++ e probabilmente sulle API di BeOS (BeAPI) per aggiungere nuovi comandi a yab. Non sono necessarie conoscenze di flex e bison.

Progettare un Comando

Un tipico comando yab o è una procedure (una funzione che ritorna void), una funzione che ritorna un numero (double o int) o una funzione che ritorna una stringa (char*).

Un comando è formato da

- le parole del comando (entità lessicali)
- la regola del comando
- la funzione di wrapping C che chiama il metodo C++
- il metodo C++

Ciò sarà discusso dettagliatamente qui di seguito.

Le parole del comando

Le parole necessarie ad un comando (tokens) sono definite nel file yabasic.flex. Per favore introdurre nuove parole del comando, solo quando quelle esistenti non sono sufficienti a descrivere i vostri nuovi comandi. Navigate nel file per trovare tutti i comandi yab e yabasic.

Una parola del comando è formata dalla parola stessa (in maiuscolo) e dal token che ritorna. Il token molto spesso è semplicemente la parola stessa con una t davanti. Ad esempio:

BUTTON ritorna tBUTTON;

Note: Ci sono eccezioni per il nome del token, es. tGET rappresenta la parola di comando GET\$ mentre tGETNUM rappresenta GET.

I nuovi token vanno dichiarati anche nel file yabasic.bison, semplicemente aggiungendo il nome del token nella lista all'inizio del file.

La regola del comando

L'attuale regola del comando (grammatica) deve essere aggiunta al file yabasic.bison. Se scorrete il file avrete una buona idea di come deve apparire la grammatica.

Fondamentalmente in questo file sono interessanti tre sezioni: la sezione per le procedure, la sezione per le funzioni numeriche e la sezione per le funzioni che ritornano una stringa. Indagheremo le differenze tra queste funzioni nelle prossime due sezioni. Ma prima diamo un'occhiata alle similitudini. Una regola del comando è scritta con un leading pipe | seguito dai token e dagli argomenti. Alla fine segue una funzione identifier. Esempio:

```
| tBUTTON coordinates to coordinates ',' string expression ',' string expression ','  
string expression {add_command(cBUTTON,NULL);}
```

Qui, coordinates è un sostituto per expression ',' expression e to è un sostituto del comando TO (che comunque è solo ',').

Così fondamentalmente ci sono due tipi di argomenti: expression è un numero (di tipo double) e string expression è una stringa (di tipo char*²). Questi possono essere separati da parentesi ', '.

Sono possibili anche ((' and ')') ma non li ho utilizzati spesso. Così in questo esempio il comando BUTTON ha 7 argomenti, 4 numeri per le coordinate e 3 stringhe che conterranno il suo ID, il testo del bottone e la

ID della view.

Procedure

Come in tutte le procedure, l'esempio menzionato no ritorna un valore (in C è una funzione void). Che da un identificatore interno chiamato cBUTTON. Questo identificatore va dichiarato nel file yabasic.h . Date un'occhiata alla lista degli altri identificatori. Inoltre, in yabasic.h va dichiarato il nome della funzione C che è stata aggiunta al file graphic.c . Le procedure danno sempre una struct di informazioni circa argomenti, linee number ecc. Cioè, in graphic.c la funzione C semplicemente trasmetterà queste informazioni alla classe principale C++ (YabInterface).

Prima di aggiungere la funzione in graphic.c essa va aggiunta anche in main.c . La, aggiungetela allo switch che chiama le funzioni in accordo al loro identificatore. Esempio:

```
case cBUTTON:
    createbutton(current, yab); DONE;
```

Una tipica funzione void in graphic.c appare come nel seguente esempio:

```
void createbutton(struct command *cmd, YabInterface* yab)
{
    double x1,y1,x2,y2;
    char *id, *title, *view;

    view = pop(stSTRING)->pointer;
    title = pop(stSTRING)->pointer;
    id = pop(stSTRING)->pointer;
    y2=pop(stNUMBER)->value;
    x2=pop(stNUMBER)->value;
    y1=pop(stNUMBER)->value;
    x1=pop(stNUMBER)->value;

    yi_SetCurrentLineNumber(cmd->line, (const char*)cmd->lib->s, yab);
    yi_CreateButton(x1,y1,x2,y2, id, title, view, yab);
}
```

Nel nostro esempio, prima i 7 argomenti yab vengono recuperati dalla struct comando. Nota: gli argomenti sono salvati sullo stack, per cui dovete recuperarli in ordine inverso! Qui, ad es. y1 viene recuperato prima di x1. Anche, le stringhe e i numeri hanno differenti pop calls. I numeri possono essere double o int ma per le coordinate, vanno utilizzati sempre i double.

La corrente linea di numeri viene passata alla classe YabInterface class chiamando yi SetCurrentLineNumber. Questa linea è la stessa per tutte le funzioni void.

Finalmente, gli argomenti vengono passati alla classe YabInterface class chiamando yi CreateButton.

Funzioni

Le funzioni che restituiscono un numero o una stringa sono implementate diversamente. Per prima cosa hanno un differente identificatore iniziale con una f es. fLISTBOXGETNUM.

Questo identificatore va dichiarato in yabasic.h nella funzione enum. Nota: le funzioni vengono ordinate in base al numero dei loro argomenti!

Inoltre il nome della funzione C che è stata aggiunta al file graphic.c va dichiarato anche in yabasic.h.

Diversamente che per le procedure,

queste funzioni impostano immediatamente il loro argomento, es.:

```
int listboxgetnum(const char*, YabInterface *yab, int line,
                 const char* libname);
```

listboxgetnum ritorna un int quando gli viene passata una stringa come primo argomento. Gli argomenti che seguono YabInterface *yab, int line, const char* libname vanno aggiunti per fornire alla classe YabInterface ulteriori informazioni. Oltre alla procedura, il recupero degli argomenti dallo stack e la chiamata alla funzione sono tutti effettuati nel file function.c. Lì gli argomenti yab vengono recuperati dallo

stack ed inoltrati alla funzione wrapper in graphic.c.

Esempio:

```
case fLISTBOXGETNUM:
    str=a1->pointer;
    value = listboxgetnum(str, yab, linenum, current->lib->s);
    result = stNUMBER;
    break;
```

L'argomento stringa è recuperato da a1->pointer. Gli argomenti numerici vengono indirizzati da es. a3->value (non presente in questo esempio). Gli argomenti vengono numerati da a1 ad a6. Al momento ulteriori argomenti non sono supportati. Qui, il risultato è archiviato come un numero. Per le stringhe, il risultato potrebbe essere archiviato in un puntatore e result = stSTRING; va impostato. Date un'occhiata agli altri comandi in questo file per ulteriori esempi. Finalmente, la funzione wrapper va implementate in graphic.c. Per il nostro esempio, ciò appare come nel seguente codice:

```
int listboxgetnum(const char* id, YabInterface *yab, int line,
                 const char* libname)
{
    yi_SetCurrentLineNumber(line, libname, yab);
    return yi_ListboxGetNum(id, yab);
}
```

La line number corrente viene passata alla classe YabInterface chiamando yi SetCurrentLineNumber. Questa linea è la stessa per tutte le funzioni. Il numero tornato viene semplicemente passato da yi ListboxGetNum.

Nota: le stringhe vanno copiate con my strdup, es. return my strdup((char*)yi CheckMessages(yab)); Spetta a voi fare in modo che le stringhe siano ancora in memoria quando sono state copiate!

La classe C++ YabInterface

- Aggiungere un Metodo

Dopo quanto descritto sopra, siamo pronti a scrivere un nuovo metodo. Il metodo ha un nome C++ e una funzione wrapper con un nome esterno che inizia con yi . Entrambe vanno definite in YabInterface.h e implementate in YabInterface.cpp.

La funzione wrapper passa il puntatore all'oggetto YabInterface e chiama il metodo principale:

```
void yi_CreateButton(double x1,double y1,double x2,double y2,
                    const char* id, const char* title,
                    const char* view, YabInterface* yab)
{
    yab->CreateButton(BRect(x1,y1,x2,y2), id, _L(title), view);
}
```

Notate la macro L() utilizzata sul testo che va tradotto automaticamente dal local kit di Zeta. Il metodo stesso è una parte della classe YabInterface che è derivata da BApplication. Così ogni metodo BApplication è accessibile dal vostro metodo.

```
void YabInterface::CreateButton(BRect frame, const char* id,
                               const char* title, const char* view)
{
    // code here
}
```

- Accesso alla strutture dati yab

yab archivia varie informazioni nelle liste degli oggetti. La lista più desiderata è la lista delle view disponibili. Queste sono archiviate nel YabList object viewList. Per inizializzare un nuovo widget, è spesso sufficiente trovare la parent view. Ciò viene fatto chiamando YabList::GetView(const char*):

```
YabView *myView = cast_as((BView*)viewList->GetView(view), YabView);
if(myView)
```

```
{
    YabWindow *w = cast_as(myView->Window(), YabWindow);
    if(w)
    {
        w->Lock();
        // initialize widget here
        w->Unlock();
    }
    else
        ErrorGen("Unable to lock window");
}
else
```

Error(view, "VIEW"); Nuovi widgets potrebbero permettere alcuni utili layout. Per favore fate riferimento ai comandi BUTTON e LISTBOX per capire come diversi tipi di layout vengono utilizzati in yab.

Se volete trovare uno specifico widget su una view sconosciuta, dovete ciclicamente attraversare le view per trovare quella che contiene il vostro widget. Un simile loop appare come qui (a condizione che MyWidget derivi da BView):

```
YabView *myView = NULL;
MyWidget *myWidget = NULL;
for(int i=0; i<viewList->CountItems(); i++)
{
    myView = cast_as((BView*)viewList->ItemAt(i), YabView);
    if(myView)
    {
        YabWindow *w = cast_as(myView->Window(), YabWindow);
        if(w)
        {
            w->Lock();
            myWidget = cast_as(myView->FindView(id), MyWidget);
            if(myWidget)
            {
                // do something with myWidget
                w->Unlock();
                return;
            }
            w->Unlock();
        }
    }
}
```

Error(id, "MyWidget"); Nota: il return è impostato dopo che qualcosa è stato fatto con il widget e dopo aver sbloccato di nuovo la finestra. Ciò consente il controllo degli errori alla fine del ciclo.

- Alcune brevi osservazioni

Alla fine alcune brevi osservazioni su...

- BUILD macros: Alcuni comandi hanno BUILD macros che consentono di disabilitare intere parti di yab durante la compilazione. Ciò viene utilizzato per produrre codice di dimensioni più contenute con build factory. Le librerie non utilizzate e parti di codice vengono semplicemente tralasciate se non necessarie. Utilizzate queste macro ogni volta che avete a che fare con comandi che li hanno.

- Drawing: I comandi di Drawing sono un pò complicati. Essi spesso disegnano su una view, su una bitmap o un canvas. Specialmente il view drawing che ha un proprio sistema di archiviazione. Osservate gli altri comandi di disegno per capire come essi lavorano.
- Own classes: Aggiungere proprie classi è bello, ma ricordate di aggiungere queste nuove informazioni anche nel makefile (R5 and ZETA!).

Sommario

Per darvi in breve una checklist di ciò che è stato fatto, date un'occhiata al sommario:

- aggiungere nuove parole di comando (token) in yabasic.flex if necessariamente
- aggiungere la regola del comando (grammar) in yabasic.bison
- aggiungere l'identificatore del comando e il nome del metodo C in yabasic.h
- aggiungere le chiamate al comando in function.c o main.c
- aggiungere la funzione wrapper in graphic.c
- aggiungere C alla funzione wrapper C++ in YabInterface.h e YabInterface.cpp
- aggiungere il metodo C++ in YabInterface.h e YabInterface.cpp

[- torna all'indice -](#)

Eventuali aggiornamenti di questo tutorial
e altra documentazione su:

WWW.Haiku-OS.it