Xcode's Plugin Interface

Abstract : this document explains how to write a plugin for integrating other compilers into Xcode, to provide a fully integrated development platform for your preferred programming language.

NEW !!! : Updated for Xcode 2.2

Table of Contents

1. Background

Apple's free IDE, Xcode, only provides support for C(++), Objective-C(++), Java, Applescript and Makefile. Although it's possible to use a Makefile for other languages, I think it's more practical to fully integrate them through dedicated plugins.

At the time I wrote this page, Xcode already has a working plugin interface (it's even used by CoreData compiler/editor, CVS/Subversion/Perforce integration, GDB debugging…). However this interface is not yet public, because not really finished. According to Apple, it'll be public in future release but no real date is provided : developers just have to wait :-(.

With a lot of reverse engineering, I've successfully understood parts of this plugin interface and started to wrote a plugin for the Objective Caml language. Because many developers would like to write Xcode plugins for various programming languages, I've decided to publish the result of my reverse engineering work.

There's actually only informations about compilers, linkers and syntax highlighting plugins.

After the OCaml plugin, I will start to write a plugin for the ADA language. So, if you are going to make an ADA plugin, please let me know in order to avoid doing the same work twice.

2. Specification Files

Xcode maintains internal arrays of many kind of objects used for every basic task. Each object (or a list of object of the same kind) is defined by a NSDictionary stored in a property list file. Xcode uses the key `Class` to let you specify which Objective-C class will be used to instantiate your object. Xcode also provide generic/default classes for each object types.

These object specifications are defined hierarchically (an object may be "BasedOn" another one, and then inherits all its properties) inside each specification kind :

- **File Type Specification** (`*.pbfilespec`) : a list of files and wrappers types defined by their extension, MIME type or magic code. You may see current available type in Xcode's preference window (section Files). Generic types are : file, text, source code, compiled code, archive, audio, image, video, folder, wrapper.
- **Language Specification** (`*.pblangspec`) : a language description, used to define syntax highlighting rules, auto-indentation rules, indexation and perhaps function name extraction (for the function popup menu), auto-completion and other editor specific tasks.
- **Compiler Specification** (`*.pbcompspec`) : a compiler description, used to describe a command line compiler (like `gcc`, `osacompile`, `javac`…), available build setting for this compiler and how to parse its output.

- **Linker Specification** (`*.pblinkspec`) : a linker description, same as compiler specifications, but for command line linkers (like `ld`, `libtool`...).
- **Product Specification** (`*.pbprodspec`) : a product is the final file created by your project (a library, an application…).
- **Package Specification** (`*.pbpackspec`) : describes the structure of a product.
- **Build Settings Specification** (`*.pbsetspec`) : defines generic (i.e. non attached to a compiler or linker) build settings accessible from the Xcode GUI.
- **Architecture Specification** (`*.pbarchspec`) : defines architectures (ppc, m68k, x86…).
- **Platform Specification** (`*.pbplatspec`) : Xcode may be used to compile code to different platforms, these specifications describe allowed architectures for each platform and where to search header files and libraries.
- **Runtime System Specification** (`*.pbRTSspec`) : available runtime system (native, java, applescript…)
- **Compile Rules** (`*.xcbuildrules`) : tell which source code file types may be compiled by a given compiler.

You may find Xcode built-in definitions here :

`/System/Library/PrivateFrameworks/DevToolsCore.framework/Resources/`

The plugin code entry points are only custom classes specified by `Class` keys in every specifications stored in it's `Contents/Resources`. If you don't use any `Class` keys, you don't need to write any code. In this case, you may just put the specification files in folder :

```
~/Library/Application Support/Apple/Developer Tools/Specifications/
```

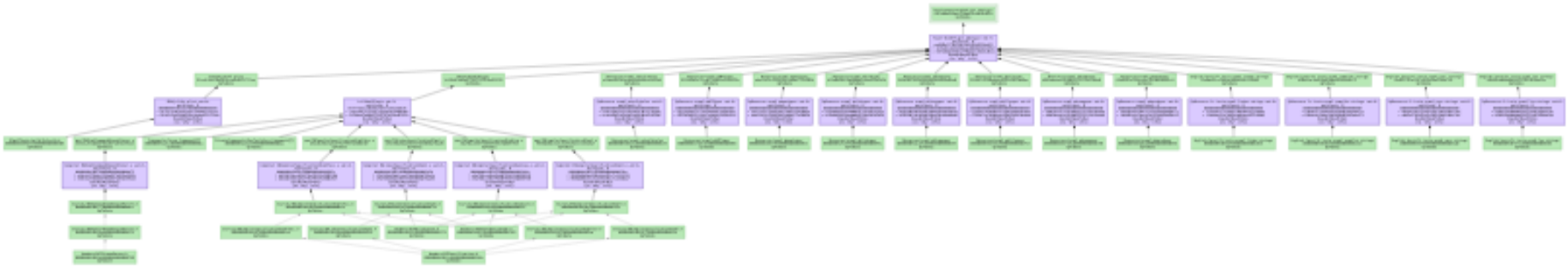
3. Dependency Graph Creation

A compiler (or linker) object is not used during the build task, but before it, when computing the dependency graph between source files, headers, compiled object files and the final product. This task is done when you open a project, add/remove a file from a target or save a edited file.

A compiler object is responsible of dependencies between source file (.c, .m…), header files (.h…) and compiled object file (.o). It also creates the command to generate the compiled object file.

A linker object is responsible of dependencies between compiled object files and the final product (an executable or library). It also generates the command used to link all object files.

When you request your project to build, Xcode just read the dependency graph and run your defined commands in the best order.



The file & command dependency graph of my OCaml Plugin created with Graphviz (click to enlarge).

4. Build Settings and Environment Variables

Like the `make` build system, Xcode use many environment variables. Theses variables are divided into three big class :

- useful paths like `$(HOME)`, `$(PRODUCT_DIR)`, `$(PROJECT_DIR)`, `$(SYSTEM_LIBRARY_DIR)`... Theses variables are defined by Xcode.
- settings from the build setting window like `$(GCC_GENERATE_DEBUGGING_SYMBOLS)`, `$(GCC_OPTIMIZATION_LEVEL)`, `$(PRODUCT_NAME)`, `$(HEADER_SEARCH_PATHS)`... The names are available in the description of each setting. Theses variables are defined in the build setting window, according to compiler and linker specifications.
- internal variables used during the dependency graph creation (in lowercase) like `$(arch)`, `$(variant)`, `$(object_files_normal_ppc)`... Theses variables are defined by your own code, or Xcode internal one's.

All variables are of type string. However, string lists may be simulated with space separated lists, as for command line tool arguments.

All Xcode strings (in build settings window, specification files, arguments of your plugin methods), may contain variable substitution (`"$(PRODUCT_DIR)/$(PRODUCT_NAME)"`) and even recursive substitution (`"$(object_files_$(variant)_$(arch))"`).

5. Building an Xcode Plugin

To create an Xcode Plugin :

- Create a new project (of type Bundle > Cocoa), or add a new target to an existing project (a target of type Cocoa > Loadable Bundle).

- modify the plugin extension to `pbplugin` in the build settings window of the target.
- add `LoadAtLaunch = YES` and `XCPluginHasUI = NO` keys to the `Info.plist` file.
- add the specification files (`*.pb*spec`) to the resources build phase.
- add the `/System/Library/PrivateFrameworks/DevToolCore.framework` framework to your project.
- add these `headers` to your project. Set the `XCODE_VERSION` macro to 22 (or 21 if you target Xcode 2.1).
- add the `Xcode plugin.pbfilespec` file (you've downloaded it with the headers) to `~/Library/Application Support/Apple/Developer Tools/Specifications/`.

A Xcode plugin must be put in the following folder to be used (do not forget to restart Xcode after) :

```
~/Library/Application Support/Apple/Developer Tools/Plug-ins/
```

Feel free to have a look at my Objective-Caml plugin source code and specification files.

6. A Word about Templates

TODO

Xcode's Plugin Interface : Specification Files

Table of Contents

1. Generic Informations about Specification Files

These files are in fact property lists, and often use the ASCII format. However, if you wish, you may use the XML or binary formats for property lists.

In each `.pb«thing»spec` file, you have an array of specifications of type «thing».

Sample file containing two specifications :

```
(
    {
        Identifier = com.domain.myxcodeplugin.spec1;
        Name = "Spec 1";              // there's a space, so quotes are required
        Class = MyOwnObjectiveCClass; // there's no space, so quotes are not required
    },
    {
        Identifier = com.domain.myxcodeplugin.spec2;
        BasedOn = com.domain.myxcodeplugin.spec1;
        «AList» = ("aaa","bbb");
        «ABoolean» = YES;    // or NO
    },
)
```

The generic properties, used in every kind of specification, are :

- `Identifier` (string) : a unique identifier, usually in inverted DNS format.
- `BasedOn` (string) : inherit all properties of the given specification.
- `Class` (string) : name of an Objective-C class used when instantiating this specification.
- `Name` (string) : a human readable short name used for Xcode's user interface.
- `Description` (string) : a description of the specification.

2. File Type Definition (`.pbfilespec`)

You'll find sample files here :

`/System/Library/PrivateFrameworks/DevToolsCore.framework/Resources/Built-in file types.pbfilespec`

`/System/Library/PrivateFrameworks/DevToolsCore.framework/Resources/Standard file types.pbfilespec`

Specific properties of file type definitions used for identification (Xcode run all these tests to find the type of a file) :

- `MIMETypes` (array of strings) : MIME types like "text/html".
- `Extensions` (array of strings) : file extensions like "mp3". "" may be used to match files without any extension.
- `TypeCodes` (array of strings or data) : Classic's four-char-code types like "TEXT" or "PICT".
- `FilenamePatterns` (array of strings) : regular expression like "[mM]akefile".
- `MagicWord` (array of strings or data) : first bytes of the file like "<?xml" or <CAFEBABE>
- `Prefix` (array of strings) : list of prefixes allowed for this kind of file
- `Permissions` (string) : set to "executable" if this kind of file must be executable.

Specific properties of file type definitions describing content :

- `ComputerLanguage` (string) : the identifier of the language used for syntax coloring (see .pblangspec files).
- `ContainsNativeCode` (bool) : set if file contains native code.
- `RequiresHardTabs` (bool) : used for files like Makefile where tabulation must not be converted to spaces.
- `IsApplication`, `IsBundle`, `IsDynamicLibrary`, `IsExecutable`, `IsExecutableWithGUI`, `IsFrameworkWrapper`, `IsStaticFrameworkWrapper`, `IsStaticLibrary`, `IsTextFile`, `IsSourceCode`, `IsFolder`, `IsWrapperFolder`, `IsProjectWrapper`, `IsDocumentation`, `IsBuildPropertiesFile` (bools) : to allow specific treatment by Xcode.
- `IsTargetWrapper` (bool) : TODO.
- `IsTransparent` (bool) : TODO.

Specific properties of file type definitions used for building :

- `AppliesToBuildRules` (bool) : can be added to the source build phase (for compilable files).
- `IsScannedForIncludes` (bool) : should Xcode search for #include or similar inclusion command ?
- `ChangesCauseDependencyGraphInvalidation` (bool) : if file is modified, it will be recompiled.
- `FallbackAutoroutingBuildPhase` (bool) : TODO.
- `IncludeInIndex` (bool) : TODO.
- `GccDialectName` : TODO.
- `CanSetIncludeInIndex` (bool) : TODO.

Specific properties of file type definitions used for wrappers :

- `ExtraPropertyNames` (dictionary) : see samples
- `ComponentParts` (dictionary) : see samples

3. Language Definition (`.pblangspec`)

You'll find a sample file here :

`/System/Library/PrivateFrameworks/DevToolsCore.framework/Resources/Built-in languages.pblangspec`

A language specification describes the syntax of a language (for syntax coloring), how it should be indent (for automatical indentation) and other things like included files or function name extracting (for the function popup).

A sample specification :

```
(
    {
        Identifier = mylang;
        Name = "My Language";
        Description = "My Language"; // may be longer
        BasedOn = "pbx_root_language";
        SourceScannerClassName = MySourceScanner;
            // (optionnal) name of an Objective-C class inheriting from PBXSourceScanner
        SupportsIndentation = NO;   // only work for C-like languages
        Indentation = {}            // indentation definition (not documented)
        SyntaxColoring = {          // syntax definition (see next paragraph)
            «prop» = «value»;
            …
        }
    }
)
```

Now, the interesting part : how to define a language syntax. The syntax is not defined by a real user grammar like in other IDEs, but by a generic grammar implemented inside Xcode, with many parameters modifiable in the language specification. Theses parameters are :

- `CaseSensitive` (bool) : is the language case sensitive ?
- `UnicodeSymbols` (bool) : does the language allow unicode characters ?
- `UnicodeEscapes` (bool) : accept \uXXXX anywhere in a file, and return a single character (not yet supported).
- `MultiLineComment` (array of two-element arrays of strings): delimiter of multiline comments like (("/*","*/"), ("(*","*)")).
- `CommentsCanBeNested` (bool) : YES if multiline comments may be nested.
- `SingleLineComment` (array of strings) : beginning delimiter of a single line comment (the comment stops at the next newline character).
- `FortranStyleComments` (bool) : allow fortran comments.
- `DocComment` (string) : characters identifying a documentation comment, just after the comment start delimiter.
- `String` (array of two-element arrays of strings) : delimiters of a string like (("\"","\"")) for "hello".
- `Character` (array of two-element arrays of strings) : delimiters of a character constant like (("'","'")) for 'h'.
- `EscapeCharacter` (string of only one character) : the escape caracter for string and character constants.
- `IdentifierStartChars` (string) : allowed characters for the first character of an identifier.
- `IdentifierChars` (string) : allowed characters for other characters of an identifier.
- `Keywords` (array of strings) : language keywords.
- `AltKeywords` (array of strings) : non official keywords.
- `DocCommentKeywords` (array of strings) : keywords in documentation comments.
- `PreprocessorKeywordStart` (string of only one character): the first caracter of all preprocessor keywords.
- `PreprocessorKeywords` (array of strings) : preprocessor keywords (without PreprocessorKeywordStart).
- `KeywordDelimeters` (not yet implemented).
- `IndexedSymbols` (bool) : TODO.
- `LinkStartChars`, `LinkPrefixChars`, `MailLocalNameDelimiter`, `DomainNameStartChars`, `DomainNameChars`, `URLSchemeDelimiter`, `URLLocationChars`, `URLSchemes` : URL coloring.

4. Installing Your own Specification File

There's two possibilities :

- if you are writing a plugin, add your specification files in `myXcodePlugin.pbplugin/Contents/Resources/`.
- otherwise, add it to `~/Library/Application Support/Apple/Developer Tools/Specifications/`.
- if you are still using Project Builder, it may have to put them in `/Developer/Project Builder Extras/Specifications`.

Xcode's Plugin Interface : Compilers & Linkers

Table of Contents

1. Xcode's Build System

Everything starts when Xcode loads your plugin bundle, just after you launch it. Xcode looks at the content of the `Info.plist` file to initialize your plugin, and read all `*.pb*spec` or `.xcspec` files of your plugin `Resources` folder. At this time, Xcode don't execute any code, except if you add a `NSPrincipalClass` entry in the `Info.plist` file, it just stores informations about what provide your plugin.

The second phase, the most important one, is the dependency graph creation. It appends every time you open a project, add/remove files to/from a project, modify some build settings… It's the phase where your plugin code is run.

Your code must create one `XCDependencyNode` for each file involved in the compilation (source file, object files, final or product files), and add dependency relations between each nodes. Some of these nodes will be marked as "product nodes" by calling `addProductNode:`, nodes representing a file of your final product. Nodes are identified by a name which is, by convention, the POSIX path of the corresponding file.

Here is the description of how is run the dependency creation phase :

- **Xcode creates a `PBXTargetBuildContext` object**. This object will be passed as parameter to all your fonctions. You must use it to modify/expand environment variables, modify the dependency graph and create commands.
- Xcode looks at the current target to know what are the product file created. All informations attached to a target are stored inside the project file. When you add a new target to a project, Xcode copies the content of the corresponding `.trgttmpl` file (in `Library/Application Support/Apple/Developer Tools/Target Templates/`) to the project.
- In the target description, there're references to all product files (files that are contained in your final product). The product files are described in `.pbprodspec` and `.pbpackspec` files. I did not really understood the differences between these two specification types, except that they are strongly related.
- **If you've created a custom `XCProductTypeSpecification` class, Xcode calls it's `computeProductDependencies…:` method** : it must contains code to define product nodes.
- Then, Xcode lists all source files (from the "source build phase" of the target). It looks at the complier rules (described in `.buildrules` files, or created by the user in the "Rules" tab of the target information window) to find which compiler must be used to compile the source file. **Xcode now call for each source file the `computeDependenciesForFilePath…` method of the appropriated compiler** (an object of a class inherited from `XCCompilerSpecification`). The code of this method must create dependency links between source and compiled file, add the compiled file to the `object_files_$(variant)_$(arch)` environment variable, create the command which will be used to compile the source file, and return the list of output nodes created by this command.
- If there're source files the returned files, Xcode automatically send it to the appropriated compiler. It may be useful for preprocessor commands like lex and yacc.
- **Now, Xcode enters the linking phase.** It first needs to know which linker to use, so it calls the `linkerSpecificationForObjectFiles…` method of the product specification object (inherited from the `XCProductTypeSpecification` class). If a linker is specified by the `compiler_mandated_linker` variable, Xcode use it instead of the linker returned by `linkerSpecificationForObjectFiles…`. Then, Xcode calls the `computeDependenciesForFilePaths…` of this linker : this method must create dependency links between product nodes and

object files, the command to be runned to call the linker, and return the list of files created by the linker.

The last phase is the real compilation/linking of files, the building phase. Here, Xcode read the dependency graph and run the command only if they are needed (i.e. if the output file doesn't exist or if the input files have been modified since last compilation) and in an best order that satisfy the dependencies.

(only for Xcode 2.1) During the building phase, Xcode create `.dot` files describing the dependency graph after and before running the commands (you'll find them in `build/«project».build/«configuration»/«target».build/`). You need the Graphviz application to view them. These .dot files are very useful to check if your code create the good dependency graph.

2. Graph Creation API

Xcode methods called by your plugin :

- Node creation (name is the file path) :
  `-[PBXTargetBuildContext dependencyNodeForName:createIfNeeded:]`
- Mark a node as "product node" :
  `-[PBXTargetBuildContext addProductNode:]`
- Create command :
  `-[PBXTargetBuildContext createCommandWithRuleInfo:commandPath:arguments:forNode:]`
- Dependency definition :
  `-[XCDependencyNode addDependedNode:]` (must be called after creating a dependency command for the output file)
  `-[XCDependencyNode addIncludeNode:]` (only for included header files)
- Add source <-> compiled file links :
  `-[PBXTargetBuildContext setCompiledFilePath:forSourceFilePath:]`

Methods of your plugin called by Xcode :

- Create product nodes :
  `-[MyProductTypeSpecification computeProductDependenciesInTargetBuildContext:]`
- Choose a linker :
  `-[MyProductTypeSpecification linkerSpecificationForObjectFilesInTargetBuildContext:]`
- Create compilation rules (called for each source file) :
  `-[MyCompilerSpecification computeDependenciesForFilePath:ofType:outputDirectory:inTargetBuildContext:]`
- Create linking rules :
  `-[MyLinkerSpecification computeDependenciesForFilePaths:outputPath:inTargetBuildContext:]`

For more informations, see the `XCPSpecifications.h`, `XCPBuildSystem.h` and `XCPDependencyGraph.h` header files available here.

3. Compiler Sample Code

Specification file `mycompiler.pbcompspec` :

```
{
    Identifier = com.domain.me.compiler;
    Class = MyCompilerSpecification;
    Name = "My Compiler";
    Description = "My source compiler";
    Version = "Default";                    // optional
    Vendor = "Me";                          // optional
    Languages = (mytype);                   // optional
```

```
    FileTypes = (sourcecode.mytype);
    ExecPath = "/usr/local/bin/mycompiler";
    CommandOutputParser = (…);
    OptionsForCommandLine = (…);           // optional (see here)
    Options = {…};                         // optional (see here)
    OptionCategories = {…};                // optional (see here)
}
```

Code file `MyCompilerSpecification.m`:

```
#import "MyCompilerSpecification.h"
#import "XCPBuildSystem.h"
#import "XCPDependencyGraph.h"

@implementation MyCompilerSpecification
- (NSArray*) computeDependenciesForInputFile:(NSString*)input ofType:(PBXFileType*)type
    variant:(NSString*)variant architecture:(NSString*)arch
    outputDirectory:(NSString*)outputDir
    inTargetBuildContext:(PBXTargetBuildContext*)context
{
    // compute input path (for variable substitution)
    input = [context expandedValueForString:input];

    // compute output path
    NSString* output = [outputDir stringByAppendingPathComponent:[[input
lastPathComponent] stringByDeletingPathExtension]];
    output = [context expandedValueForString:output];

    // create dependency nodes
    XCDependencyNode* outputNode = [context dependencyNodeForName:output createIfNeeded:
YES];
    XCDependencyNode* inputNode = [context dependencyNodeForName:input createIfNeeded:YES];

    // create compiler command
    XCDependencyCommand* dep = [context
        createCommandWithRuleInfo:[NSArray arrayWithObjects:@"MyCompile",[context
naturalPathForPath:input],nil]
        commandPath:[context expandedValueForString:[self path]]
        arguments:nil
        forNode:outputNode];
    [dep setToolSpecification:self];
    [dep addArgumentsFromArray:[self
commandLineForAutogeneratedOptionsInTargetBuildContext:context]];
    [dep addArgument:@"-o"];
    [dep addArgument:output];
    [dep addArgument:input];

    // create dependency rules
    [outputNode addDependedNode:inputNode];

    // update source-compiled links
    [context setCompiledFilePath:output forSourceFilePath:input];

    // add to the list of file for the linker
    NSString* object_files_variant_arch = [context expandedValueForString:@"object_files_
$(variant)_$(arch)"];
    [context appendStringOrStringListValue:output toDynamicSetting:
object_files_variant_arch];
```

```
    // return output object node
    return [NSArray arrayWithObject:outputNode];
}
@end
```

4. Linker Sample Code

Specification file `mylinker.pblinkspec` :

```
{
    Identifier = com.domain.me.linker;
    Class = MyLinkerSpecification;
    Name = "My linker";
    Description = "My linker";
    Version = "Default";                    // optional
    Vendor = "Me";                          // optional
    BinaryFormats = ("mach-o");
    Architectures = (ppc);
    ExecPath = "/usr/local/bin/mylinker";
    InputFileTypes = (compiled.mach-o.objfile);
    CommandOutputParser = (…);
    OptionsForCommandLine = (…);            // optional (see here)
    Options = {…};                          // optional (see here)
    OptionCategories = {…};                 // optional (see here)
}
```

Code file `MyLinkerSpecification.m` :

```
#import "MyLinkerSpecification.h"
#import "XCPBuildSystem.h"
#import "XCPDependencyGraph.h"

@implementation MyLinkerSpecification
- (NSArray*)computeDependenciesForFilePaths:(NSArray*)inputs
    outputPath:(NSString*)output
    inTargetBuildContext:(PBXTargetBuildContext*)context
{
    // compute output path (for variable substitution)
    output = [context expandedValueForString:output];

    // create linker command
    XCDependencyNode* outputNode = [context dependencyNodeForName:output createIfNeeded:
YES];
    XCDependencyCommand* dep = [context
        createCommandWithRuleInfo:[NSArray arrayWithObjects:@"MyLink",[context
naturalPathForPath:output],nil]
        commandPath:[context expandedValueForString:[self path]]
        arguments:nil
        forNode:outputNode];
    [dep setToolSpecification:self];
    [dep addArgumentsFromArray:[self
commandLineForAutogeneratedOptionsInTargetBuildContext:context]];
    [dep addArgument:@"-o"];
    [dep addArgument:output];
```

```
    // some types
    PBXFileType* myObjectFileType = [PBXFileType specificationForIdentifier:@"compiled.
myobjfile"];
    PBXFileType* myLibraryFileType = [PBXFileType specificationForIdentifier:@"compiled.
mylibraryfile"];

    // create dependency rules & command arguments for libraries
    NSEnumerator* libraryEnum = [[context linkedLibraryPaths] objectEnumerator];
    NSString* library;
    while((library = [libraryEnum nextObject]) != nil) {
        library = [context expandedValueForString:library];
        PBXFileType* type = [PBXFileType fileTypeForFileName:[library lastPathComponent]];

        if([type isKindOfSpecification:myLibraryFileType] || [type isKindOfSpecification:
myObjectFileType]) {
            XCDependencyNode* libraryNode = [context dependencyNodeForName:library
createIfNeeded:YES];
            [outputNode addDependedNode:libraryNode];
            [dep addArgument:library];

        } else {
            [context addDependencyAnalysisWarningMessageFormat:
                @"warning: skipping file '%@' (unexpected file type '%@' in Frameworks &
Libraries build phase)",
                library, [type identifier]];
        }
    }

    // create dependency rules & command arguments for compiled object
    NSEnumerator* objectEnum = [inputs objectEnumerator];
    NSString* input;
    while((input = [objectEnum nextObject]) != nil) {
        input = [context expandedValueForString:input];
        PBXFileType* type = [PBXFileType fileTypeForFileName:[input lastPathComponent]];

        if([type isKindOfSpecification:myObjectFileType]) {
            XCDependencyNode* inputNode = [context dependencyNodeForName:input
createIfNeeded:YES];
            [outputNode addDependedNode:inputNode];
            [dep addArgument:input];

        } else {
            [context addDependencyAnalysisWarningMessageFormat:
                @"warning: skipping file '%@' (unexpected file type '%@' in link phase)",
                input, [type identifier]];
        }
    }

    return [NSArray arrayWithObject:outputNode];
}
@end
```

5. Product Type Sample Code

Specification file `myproducttype.pbprodspec` :

```
{
```

```
    Identifier = com.domain.me.product-type.tool;
    Class = MyProductTypeSpecification;
    Name = "My Command-line Tool";
    Description = "My Command-line Tool";
    //Image name for inactive target is <IconNamePrefix>
    //Image name for active target is <IconNamePrefix>Active
    IconNamePrefix = "TargetPlugin";
    DefaultTargetName = "MyTool";
    DefaultBuildProperties = { … };
    AllowedBuildPhaseTypes = ( Headers, Sources, Frameworks );
    PackageTypes = (
        com.domain.me.package-type.tool
    );
}
```

Code file `MyProductTypeSpecification.m` :

```
#import "MyProductTypeSpecification.h"
#import "XCPBuildSystem.h"

@implementation MyProductTypeSpecification
- (void)computeProductDependenciesInTargetBuildContext:(PBXTargetBuildContext*)context
{
    NSString* productPath = [context expandedValueForString:@"$(TARGET_BUILD_DIR)/
$(EXECUTABLE_PATH)"];
    XCDependencyNode* productNode = [context dependencyNodeForName:productPath
createIfNeeded:YES];
    [context addProductNode:productNode];
}

- (void)computePostprocessingDependenciesInTargetBuildContext:(PBXTargetBuildContext*)
context
{
}

- (XCLinkerSpecification*)linkerSpecificationForObjectFilesInTargetBuildContext:
(PBXTargetBuildContext*)context
{
    return [[XCLinkerSpecification specificationRegistry] objectForKey:@"com.domain.me.
linker"];
}
@end
```

Xcode's Plugin Interface : Build Settings

You'll find some official informations about build setting variables here :

- Apple's chapter about the build setting system
- Build Setting Evaluation
- Some build setting variables and their short description

Table of Contents

1. Adding Build Settings to a Compiler/Linker

TODO

2. Environment API

Expand variables :

- Expand as a string : `[PBXBuildSetting expandedValueForString:]`
- Expand as a path : `[PBXBuildSetting absoluteExpandedPathForString:]`
- Expand as a boolean value : `[PBXBuildSetting expandedBooleanValueForString:]` (Xcode 2.2 only)
- Find if there is a variable to expand : `[PBXBuildSetting expandedValueIsNonEmptyForString:]`
- ~~Find if there is a variable to expand : `[PBXBuildSetting expandedValueExistsForString:]`~~ (Xcode 2.1 only)
- ~~Expand a string : `[PBXBuildSetting expandedValueForString:getRecursiveSettingName:options:]` (the two last parameters should be respectively `nil` and `1`)~~ (Xcode 2.1 only)

Modify string variables :

- Set : `[PBXBuildSetting setStringValue:forDynamicSetting:]`
- Delete : `[PBXBuildSetting removeDynamicSetting:]`

Modify string list variables :

- Insert a string at the beginnig : `[PBXBuildSetting prependStringOrStringListValue:toDynamicSetting:]`
- Insert a string at the end : `[PBXBuildSetting appendStringOrStringListValue:toDynamicSetting:]`
- Remove an string from the list : `[PBXBuildSetting removeStringOrStringListValue:]`

Some useful tasks :

- Convert a string to a boolean value : `[NSString boolValue]`
- Convert a space separated string list to a string array : `[NSString arrayByParsingAsStringList]`

Compute command line arguments from build settings :

- Defined in the `Options` key : `[XCCommandLineToolSpecification commandLineForAutogeneratedOptionsForKey: inTargetBuildContext:]`
- Defined in a custom key :

```
[XCCommandLineToolSpecification commandLineForAutogeneratedOptionsInTargetBuildContext:]
```

For more informations, see the `XCPBuildSystem.h`n `XCPSpecifications.h` and `XCPSupport.h` header files available here.

3. Useful Environment Variables

In many case, it's possible to provide different values when compiling to different architectures or different variant : `MYSETTING_ppc`, `MYSETTING_x86`, `MYSETTING_normal`, `MYSETTING_debug`, `MYSETTING_normal_ppc`.

Variables containing useful paths :

- Local applications folder : `LOCAL_APPS_DIR`
- Local utilities folder : `LOCAL_ADMIN_APPS_DIR`
- Local library folder : `LOCAL_LIBRARY_DIR`
- User folder : `HOME`
- User applications folder : `USER_APPS_DIR`
- User library folder : `USER_LIBRARY_DIR`
- Project base : `PROJECT_DIR`
- Project output folder (for final product) : `TARGET_BUILD_DIR` or `BUILT_PRODUCTS_DIR`
- Project objects folder (for .o files) : `OBJECTS_DIR`
- Project derived files folder (for yacc, lex output) : `DERIVED_FILE_DIR`
- Temporary folder : `TEMP_DIR`

Variables used to defined the product :

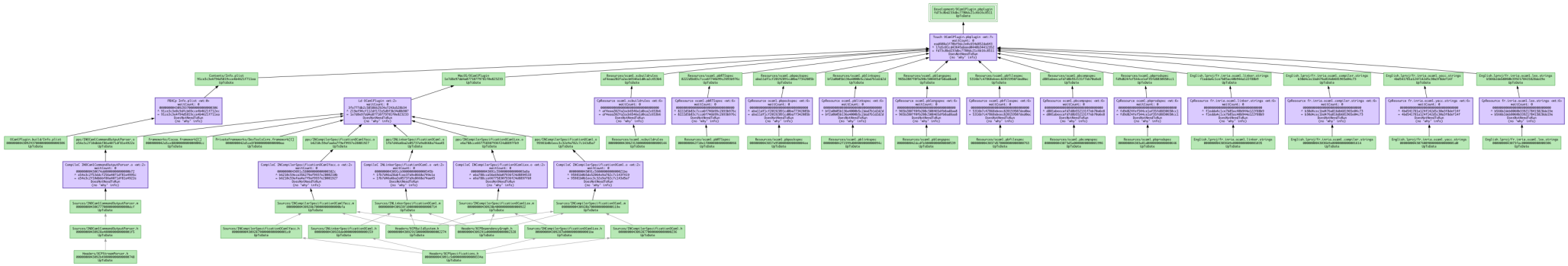
- Installation path : `INSTALL_PATH`. A good value is : `$(HOME)/bin` for command line tools, `$(HOME)/Applications` for applications, `$(HOME)/Library/Frameworks` for frameworks…
- Installation path for headers : `PUBLIC_HEADERS_FOLDER_PATH`
- Product name : `PRODUCT_NAME`
- Project file name : `PROJECT_NAME`
- Executable name : `EXECUTABLE_NAME`. A good value is : `$(EXECUTABLE_PREFIX)$(PRODUCT_NAME)$(EXECUTABLE_VARIANT_SUFFIX)$(EXECUTABLE_SUFFIX)`.
- Architectures : `ARCHS`
- Build variants : `BUILD_VARIANTS`

Variables containing compilation/linking directives :

- Header paths : `HEADER_SEARCH_PATHS`
- Library paths : `LIBRARY_SEARCH_PATHS`
- Framework paths : `FRAMEWORK_SEARCH_PATHS`
- Should generate profiling code : `GENERATE_PROFILING_CODE`
- Type of libary : `LIBRARY_STYLE` (= `STATIC`, `DYNAMIC` or `BUNDLE`)

Variables used during dependency graph creation (lowercase) :

- Current variant being processed : `variant`
- Current architecture being processed : `arch`
- Object files sent to the linker : `object_files_$(variant)_$(arch)`
- ~~Force to use the given linker : `compiler_mandated_linker`~~ (use a custom product specification instead)


/Developement/XCodePlugin.pbplugin
0df7c8bd218abc7f3bba3fca6d1bcdf21
0pfsdata
Touch XCodePlugin.pbplugin -set-?-
mutCount: 0
xcpbUKuif7Mor0qc3eEo4qRS1dadd1
= 0df7c8bd218abc7f3bba3fca6d1bcdf21
: 0df7c8bd218abc7f3bba3fca6d1bcdf21
DoesNotNeedToRun
{No 'Why' Info}
Contents/Info.plist
91ca4c3ae0ea0ab4ab4a4a9d1f753ab
0pfsdata
MacOS/XCodeFlugin
1e7d8e97abcaa8ab4a7a7f1ab4d3233
0pfsdata
Resources/xcool.scbuildrules
afdeaa3b1f1ae1ab4ab1ab4ab1ab4b3b6
0pfsdata
Resources/xcool.pbXTSspec
82224bd5c7cc2ab4ab4a4a9c1e9f8d4fa
0pfsdata
Resources/xcool.pbpackspec
dea41af1cf7d6d3f1ab4ab4a4a9c77e2b89a
0pfsdata
Resources/xcool.pblinkspec
bf1a6b4bf1ab4ab4a4a9c1aaaf5cb42d3
0pfsdata
Resources/xcool.pbtargspec
9f03c2b6f9bf1ab4ab4a4a9c1aaaf4baa8aad
0pfsdata
Resources/xcool.pbfilespec
b1cbdc1c478da4ab4a4a9c1aaaf5c9ea8ad
0pfsdata
Resources/xcool.pbcompspec
d8d1aaaaccaf1ab4ab4a4a9c1aaaf5c9ea8
0pfsdata
Resources/xcool.pbprodspec
f2b4b3cf1ab4ab4a4a9c1aaaf5c9ea8ac1
0pfsdata
English.lproj/fr.ioria.xcool.linker.strings
f1a6aa6c1ab4ab4a4a9c1aaaf5c9ea8a1227d8d4
0pfsdata
English.lproj/fr.ioria.xcool.compiler.strings
b4b6acaa1ab4ab4a4a9c1aaaf5c9ea8a1c75
0pfsdata
English.lproj/fr.ioria.xcool.yacc.strings
6bab1781a1ab4ab4a4a9c1aaaf5c9ea8a1ab
0pfsdata
English.lproj/fr.ioria.xcool.lex.strings
b5d6b2ab4ab4a4a9c1aaaf5c9ea8a1b2b2a2be
0pfsdata
PkgCp Info.plist -set-0c-
mutCount: 0
0000000e1d42c2170000000000002d6c
= 0000000e1d42c2170000000000002d6c
= 91ca13ef9fe6d2d2c1ce04e82771f1ae
DoesNotNeedToRun
{No 'Why' Info}
Ld XCodeFlugin -set-2-
mutCount: 0
31c777dcc5d5bb2b2c4e7241c2b1b34
= 31c777dcc5d5bb2b2c4e7241c2b1b34
: 2e7d8e97abcaf73bf75f67c9ad13233
DoesNotNeedToRun
{No 'Why' Info}
CpResource xcool.scbuildrules -set-0c-
mutCount: 0
0000000000000000000000000000000000000000
= afdeaa3b1f1ae1ab4ab1ab4ab1ab4b3b6
DoesNotNeedToRun
{No 'Why' Info}
CpResource xcool.pbXTSspec -set-6c-
mutCount: 0
0000000000000000000000000000000000000000
= 82224bd5c7cc2ab4ab4a4a9c1e9f8d4fa
DoesNotNeedToRun
{No 'Why' Info}
CpResource xcool.pbpackspec -set-6c-
mutCount: 0
0000000000000000000000000000000000000000
= dea41af1cf7d6d3f1ab4ab4a4a9c77e2b89a
DoesNotNeedToRun
{No 'Why' Info}
CpResource xcool.pblinkspec -set-6c-
mutCount: 0
0000000000000000000000000000000000000000
= bf1a6b4bf1ab4ab4a4a9c1aaaf5cb42d3
DoesNotNeedToRun
{No 'Why' Info}
CpResource xcool.pbtargspec -set-6c-
mutCount: 0
0000000000000000000000000000000000000000
= 9f03c2b6f9bf1ab4ab4a4a9c1aaaf4baa8aad
DoesNotNeedToRun
{No 'Why' Info}
CpResource xcool.pbfilespec -set-6c-
mutCount: 0
0000000000000000000000000000000000000000
= b1cbdc1c478da4ab4a4a9c1aaaf5c9ea8ad
DoesNotNeedToRun
{No 'Why' Info}
CpResource xcool.pbcompspec -set-6c-
mutCount: 0
0000000000000000000000000000000000000000
= d8d1aaaaccaf1ab4ab4a4a9c1aaaf5c9ea8
DoesNotNeedToRun
{No 'Why' Info}
CpResource xcool.pbprodspec -set-6c-
mutCount: 0
0000000000000000000000000000000000000000
= f2b4b3cf1ab4ab4a4a9c1aaaf5c9ea8ac1
DoesNotNeedToRun
{No 'Why' Info}
CpResource fr.ioria.xcool.linker.strings -set-6c-
mutCount: 0
0000000000000000000000000000000000000000
= f1a6aa6c1ab4ab4a4a9c1aaaf5c9ea8a1227d8d4
DoesNotNeedToRun
{No 'Why' Info}
CpResource fr.ioria.xcool.compiler.strings -set-6c-
mutCount: 0
0000000000000000000000000000000000000000
= b4b6acaa1ab4ab4a4a9c1aaaf5c9ea8a1c75
DoesNotNeedToRun
{No 'Why' Info}
CpResource fr.ioria.xcool.yacc.strings -set-6c-
mutCount: 0
0000000000000000000000000000000000000000
= 6bab1781a1ab4ab4a4a9c1aaaf5c9ea8a1ab
DoesNotNeedToRun
{No 'Why' Info}
CpResource fr.ioria.xcool.lex.strings -set-6c-
mutCount: 0
0000000000000000000000000000000000000000
= b5d6b2ab4ab4a4a9c1aaaf5c9ea8a1b2b2a2be
DoesNotNeedToRun
{No 'Why' Info}
XCodeFlugin.build/Info.plist
0000000e1d42c2170000000000002d6c
0pfsdata
ppc/INXCodeCommandOutputParser.o
e9dcc7cd7ab4ab4a4a9c1aaaf5cab4a2a
0pfsdata
Frameworks/Cocoa.framework/C[]
0000000e1cab4ab4a4a9c1aaaf5c9ea8
0pfsdata
PrincipalFrameworks/SecFoundCore.framework/C[]
0000000e1cab4ab4a4a9c1aaaf5c9ea8
0pfsdata
ppc/INCompilerSpecificationXCodeFlex.o
bdc18c39ae3e0000000c7e2b8d81927
0pfsdata
ppc/INLinkerSpecificationXCodeCmd.o
b7b7e0ea0ab4ab4a4a9c1aaaf5c9ea8ab4aa6
0pfsdata
ppc/INCompilerSpecificationXCodeCmd.o
e4ad3bcca0ab4ab4a4a9c1aaaf5c9ea8ab4
0pfsdata
ppc/INCompilerSpecificationXCodeCmd.o
9956ab4ab1ab4ab4a4a9c1aaaf5c1a3a8a2
0pfsdata
Resources/xcool.xcbuildrules
afdeaa3b1f1ae1ab4ab1ab4ab1ab4b3b6
0pfsdata
Resources/xcool.pbXTSspec
82224bd5c7cc2ab4ab4a4a9c1e9f8d4fa
0pfsdata
Resources/xcool.pbpackspec
dea41af1cf7d6d3f1ab4ab4a4a9c77e2b89a
0pfsdata
Resources/xcool.pblinkspec
bf1a6b4bf1ab4ab4a4a9c1aaaf5cb42d3
0pfsdata
Resources/xcool.pbtargspec
9f03c2b6f9bf1ab4ab4a4a9c1aaaf4baa8aad
0pfsdata
Resources/xcool.pbfilespec
b1cbdc1c478da4ab4a4a9c1aaaf5c9ea8ad
0pfsdata
Resources/xcool.pbcompspec
d8d1aaaaccaf1ab4ab4a4a9c1aaaf5c9ea8
0pfsdata
Resources/xcool.pbprodspec
f2b4b3cf1ab4ab4a4a9c1aaaf5c9ea8ac1
0pfsdata
English.lproj/fr.ioria.xcool.linker.strings
f1a6aa6c1ab4ab4a4a9c1aaaf5c9ea8a1227d8d4
0pfsdata
English.lproj/fr.ioria.xcool.compiler.strings
b4b6acaa1ab4ab4a4a9c1aaaf5c9ea8a1c75
0pfsdata
English.lproj/fr.ioria.xcool.yacc.strings
6bab1781a1ab4ab4a4a9c1aaaf5c9ea8a1ab
0pfsdata
English.lproj/fr.ioria.xcool.lex.strings
b5d6b2ab4ab4a4a9c1aaaf5c9ea8a1b2b2a2be
0pfsdata
CompileC INCommandOutputParser.o -set-2-
mutCount: 0
0000000e1d67a1b50000000000003d72
= 0000000e1d67a1b50000000000003d72
: d54cfc7f3b8bbbf80a4ff1df61a4922c
DoesNotNeedToRun
{No 'Why' Info}
CompileC INCompilerSpecificationXCodeFlex.o -set-2-
mutCount: 0
0000000e1d67a1b50000000000003d72
= bdc18c39ae3e0000000c7e2b8d81927
: c19c7d40ad0ab5df5f3fe8b08dc9aae6
DoesNotNeedToRun
{No 'Why' Info}
CompileC INLinkerSpecificationXCodeCmd.o -set-2-
mutCount: 0
0000000e1d67a1b50000000000003e5c
= b7b7e0ea0ab4ab4a4a9c1aaaf5c9ea8ab4aa6
: c19c7d40ad0ab5df5f3fe8b08dc9aae6
DoesNotNeedToRun
{No 'Why' Info}
CompileC INCompilerSpecificationXCodeCmd.o -set-2-
mutCount: 0
0000000e1d67a1b50000000000003e5c
= e4ad3bcca0ab4ab4a4a9c1aaaf5c9ea8ab4
: 9956ab4ab1ab4ab4a4a9c1aaaf5c1a3a8a2
DoesNotNeedToRun
{No 'Why' Info}
CompileC INCompilerSpecificationXCodeCmd.o -set-2-
mutCount: 0
0000000e1d67a1b50000000000003e5c
= 9956ab4ab1ab4ab4a4a9c1aaaf5c1a3a8a2
: 9956ab4ab1ab4ab4a4a9c1aaaf5c1a3a8a2
DoesNotNeedToRun
{No 'Why' Info}
Sources/INCompilerSpecificationXCodeFlex.m
0000000e1d67a1b50000000000003e5c
0pfsdata
Sources/INLinkerSpecificationXCodeCmd.m
0000000e1d67a1b50000000000003f3d
0pfsdata
Sources/INCompilerSpecificationXCodeCmdLex.m
0000000e1d67a1b50000000000004012
0pfsdata
Sources/INCompilerSpecificationXCodeCmd.m
0000000e1d67a1b50000000000004134
0pfsdata
Sources/INCompilerSpecificationXCodeFlex.h
0000000e1d67a1b7b0000000000003ce8
0pfsdata
Sources/INLinkerSpecificationXCodeCmd.h
0000000e1d67a1ba0000000000003c55
0pfsdata
Headers/XCPBuildSystem.h
0000000e1d67a1c1a0000000000003279
0pfsdata
Headers/XCPDependencyGraph.h
0000000e1d67a1c1a0000000000003c2a8
0pfsdata
Sources/INCompilerSpecificationXCodeCmdLex.h
0000000e1d67a7b0000000000003d1ba
0pfsdata
Sources/INCompilerSpecificationXCodeCmd.h
0000000e1d67a1c1a0000000000003c26
0pfsdata
Headers/XCPFoundationCore.h
0000000e1d67a1a0000000000003d7dd
0pfsdata
Headers/XCPSpecificationCore.h
0000000e1d67a1a0000000000003c34a
0pfsdata

Objective Caml Plugin for Xcode

NEW !! : now compatible with Xcode 2.2

Current version : beta 5. Download it

Table of Contents

1. Background

Xcode only provide support for C(++), Objective-C(++), Java, Applescript and Makefile. Even if it's possible to use a makefile for other languages, I think it's more practical to have a fully integrated them throught plugins.

At the time I wrote this page, Xcode has a fully working plugin interface (it's even use by CoreData compiler, CVS/Subversion/Perforce integration…). However this interface is not yet public, and, according to Apple, it'll be public in futur release but no real date is provided : developers just have to wait :(. With a lot of reverse engineering, I've successfully understand parts of this plugin interface and started to wrote my OCaml plugin. Parts of this reverse engineering work are published on my site.

Objective-Caml is a programming langage created by the INRIA, a french research center in computer science. For more informations, go to the Objective Caml web site. Extract from this site :

> *Objective Caml is the most popular variant of the Caml language. From a language standpoint, it extends the core Caml language with a fully-fledged object-oriented layer, as well as a powerful module system, all connected by a sound, polymorphic type system featuring type inference.*
>
> *The Objective Caml system is an industrial-strength implementation of this language, featuring a high-performance native-code compiler (ocamlopt) for 9 processor architectures (IA32, PowerPC, AMD64, Alpha, Sparc, Mips, IA64, HPPA, StrongArm), as well as a bytecode compiler (ocamlc) and an interactive read-eval-print loop (ocaml)*

for quick development and portability. The Objective Caml distribution includes a comprehensive standard library, a replay debugger (ocamldebug), lexer (ocamllex) and parser (ocamlyacc) generators, a pre-processor pretty-printer (camlp4) and a documentation generator (ocamldoc).

2. Description

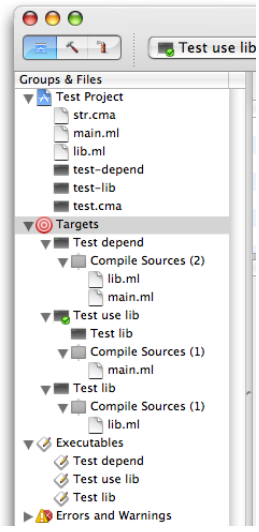
This Xcode plugin allow you to :

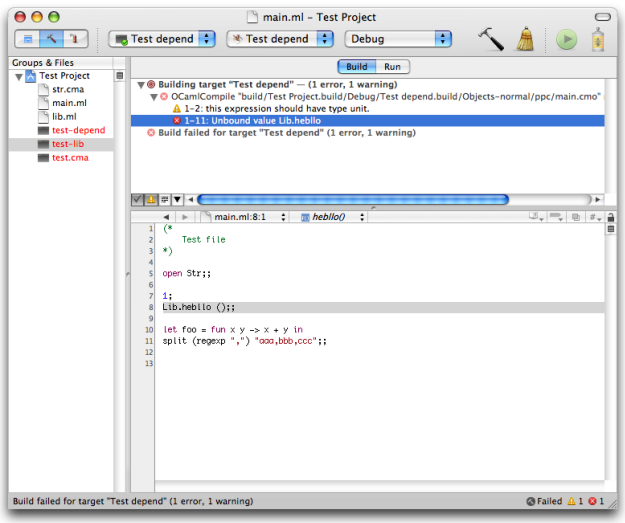
- create a Objective Caml projet or target with Xcode
- add files to this target like you do for a C program
- modify the OCaml compiler or linker settings through the Xcode's build-in build setting GUI.
- adding a library in library build phase of a OCaml target
- syntax higlighting
- use ocamllex and ocamlyacc
- use of the native compiler (select architecture "ppc" instead of the default "ocaml")

However, as it's a beta version, it doesn't support yet :

- dependance between source files (just must put them manually in the good order)
- run the complied program from Xcode (you must use the terminal)
- ocamlp4 preprocessor
- mixing OCaml and C code
- debugging from Xcode (but possible with direct use of ocamldebug)

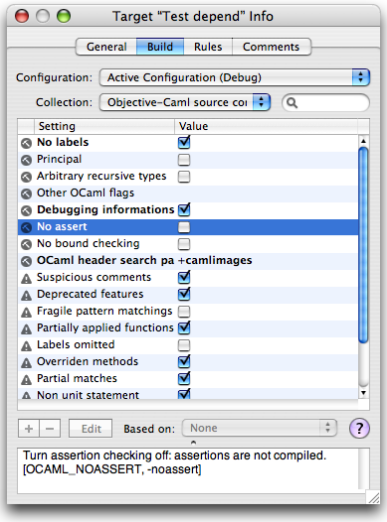
3. Screen shots (click to enlarge)

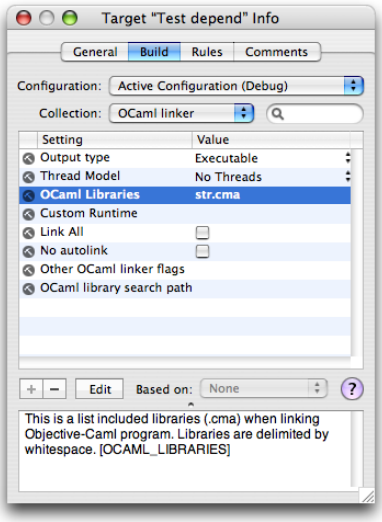


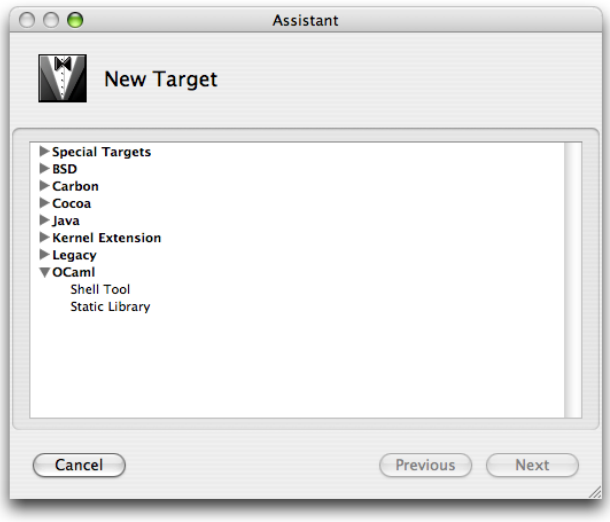


Groups & Files

Errors & warnings, syntax coloration







Compiler build Settings

Linker build Settings

OCaml targets

4. Download & Install

Requirement :

- Xcode 2.2 (which itself run only on Mac OS X 10.4.x).
- Objective Caml development tools 3.08.3 or later (available here).

To install the plugin, you have to put the following files in your Library folder or in the system Library folder :

- `OCamlPlugin.pbplugin` in :
  `Library/Application Support/Apple/Developer Tools/Plug-ins/`
- Target templates in :
  `Library/Application Support/Apple/Developer Tools/Target Templates/OCaml/`
- Project templates in :
  `Library/Application Support/Apple/Developer Tools/Project Templates/OCaml/`

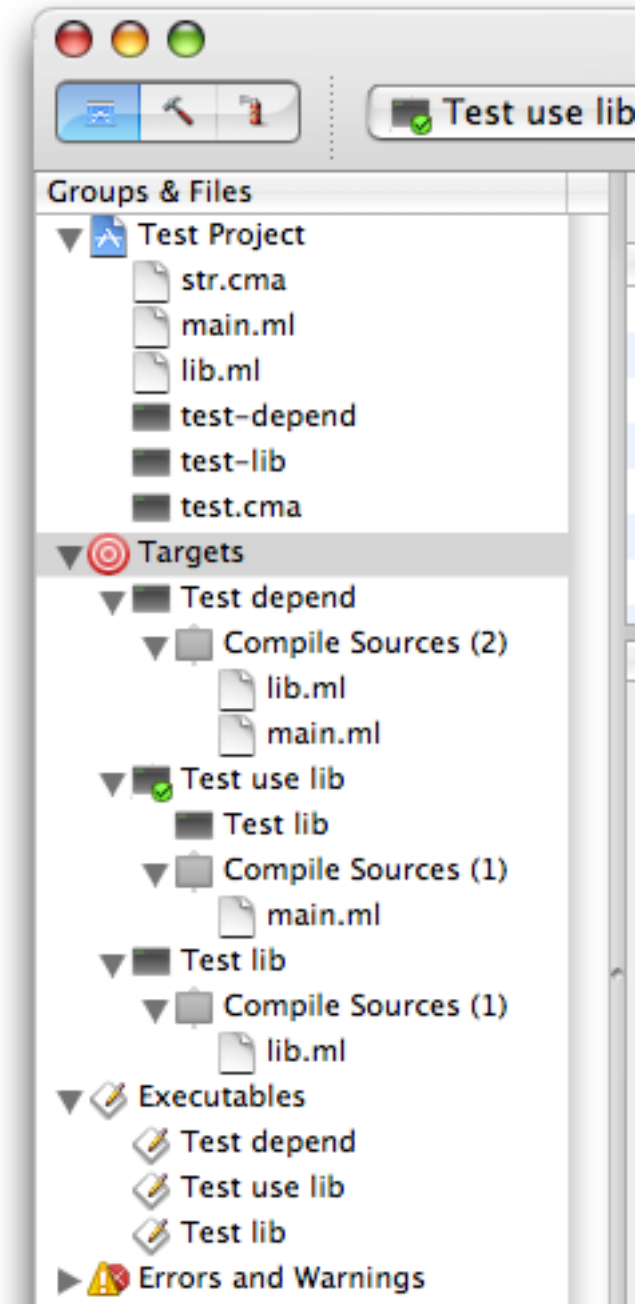
5. Some Current Tasks

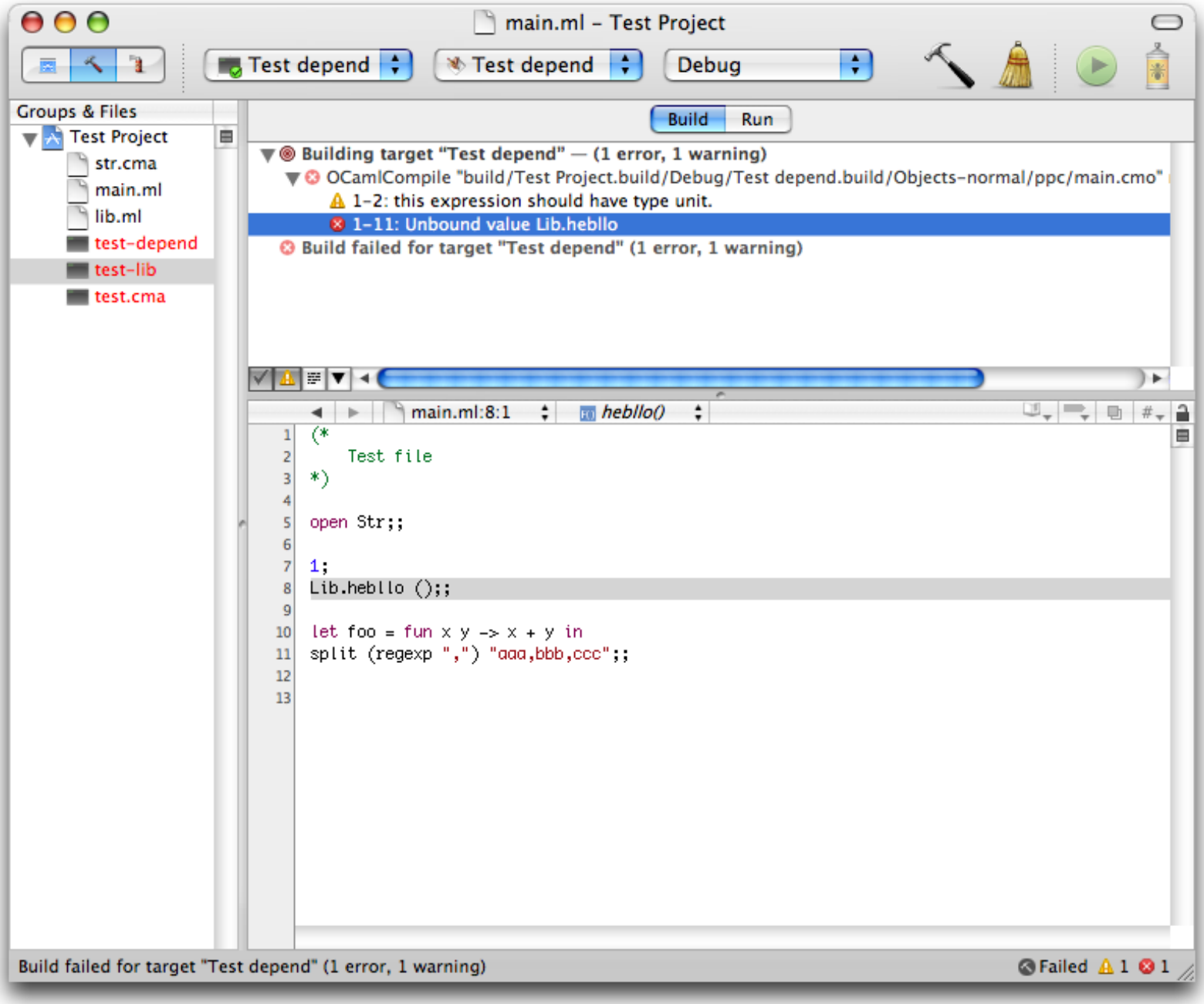
TODO

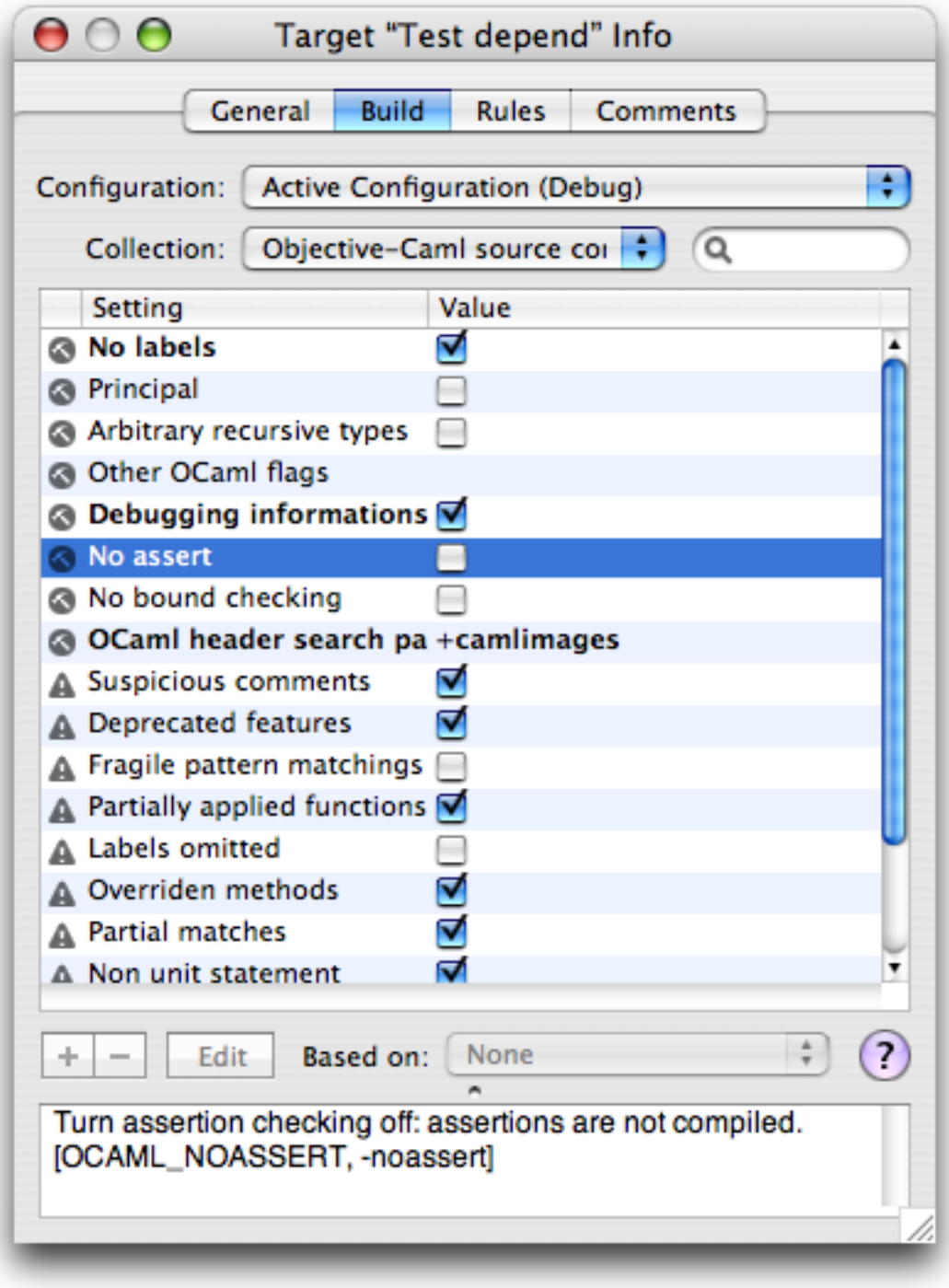
6. Source Code

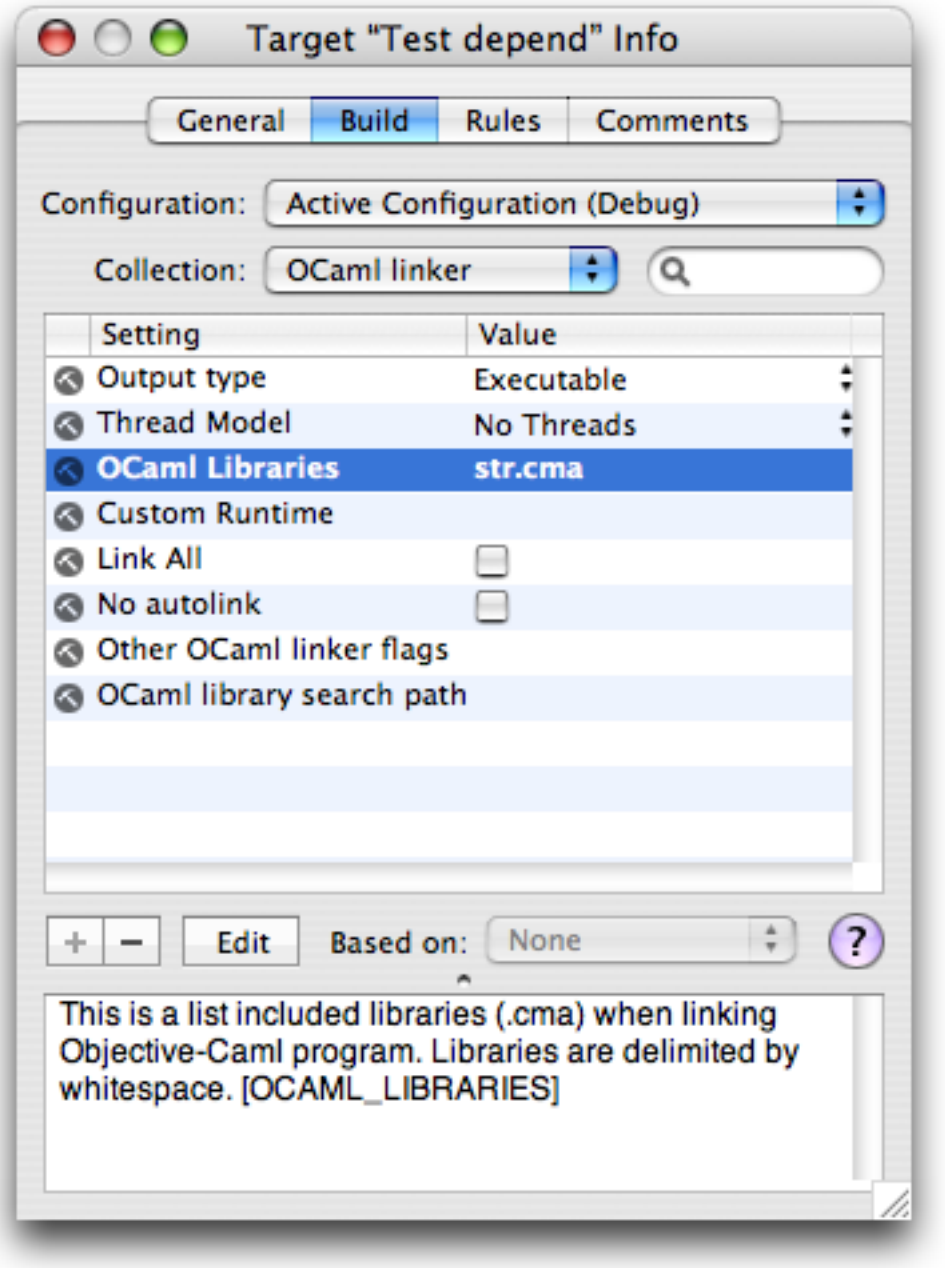
The source code is distributed under the GPL license.

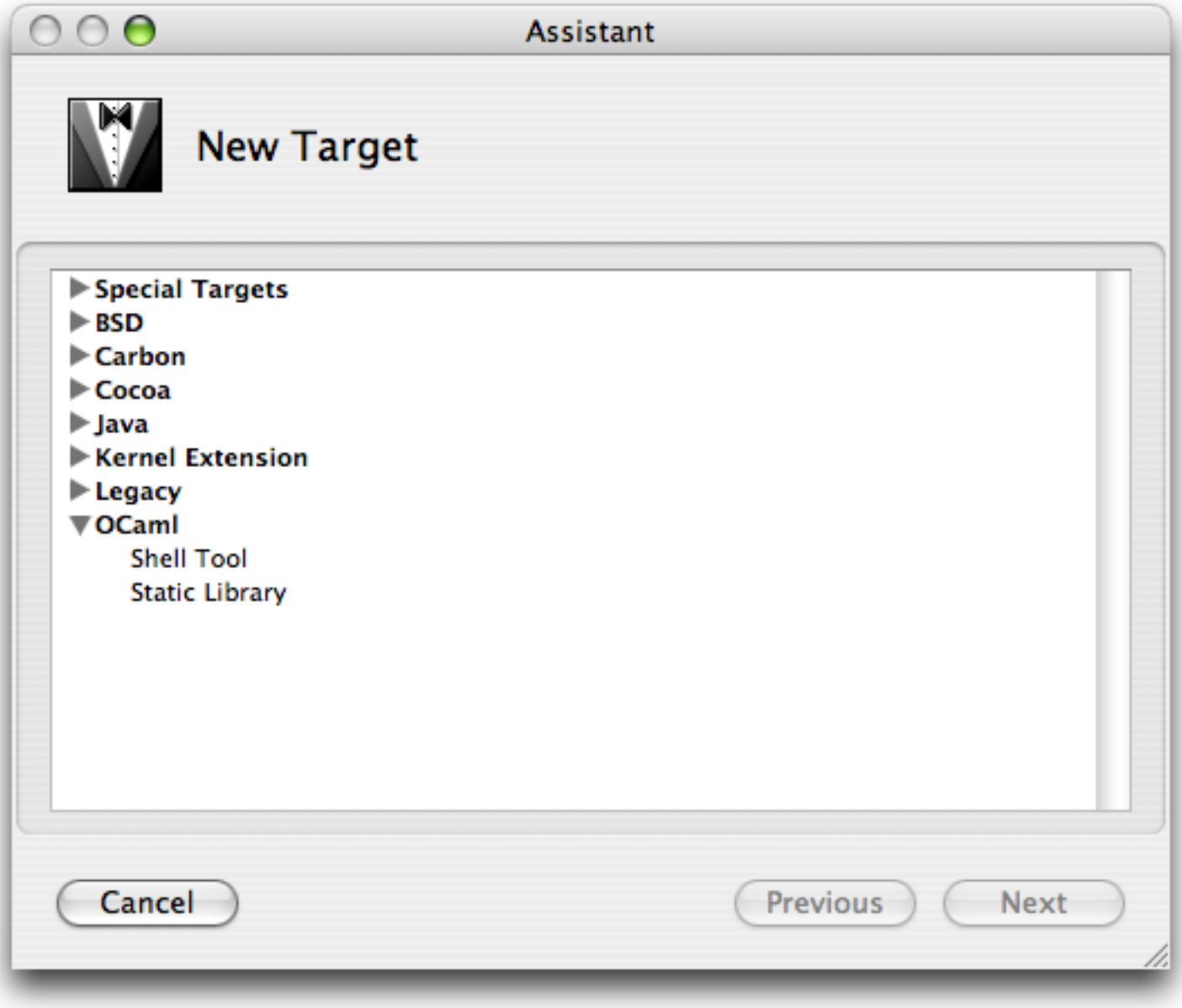
Download the source code.


Test use lib
Groups & Files
Test Project
str.cma
main.ml
lib.ml
test-depend
test-lib
test.cma
Targets
Test depend
Compile Sources (2)
lib.ml
main.ml
Test use lib
Test lib
Compile Sources (1)
main.ml
Test lib
Compile Sources (1)
lib.ml
Executables
Test depend
Test use lib
Test lib
Errors and Warnings


main.ml – Test Project
Test depend
Test depend
Debug
Groups & Files
Test Project
str.cma
main.ml
lib.ml
test-depend
test-lib
test.cma
Build
Run
Building target "Test depend" — (1 error, 1 warning)
OCamlCompile "build/Test Project.build/Debug/Test depend.build/Objects-normal/ppc/main.cmo"
1-2: this expression should have type unit.
1-11: Unbound value Lib.hebllo
Build failed for target "Test depend" (1 error, 1 warning)
main.ml:8:1
hebllo()
(*
Test file
*)
open Str;;
1;
Lib.hebllo ();;
let foo = fun x y -> x + y in
split (regexp ",") "aaa,bbb,ccc";;
Build failed for target "Test depend" (1 error, 1 warning)
Failed 1 1


Target "Test depend" Info
General
Build
Rules
Comments
Configuration:
Active Configuration (Debug)
Collection:
Objective-Caml source co
Setting
Value
No labels
Principal
Arbitrary recursive types
Other OCaml flags
Debugging informations
No assert
No bound checking
OCaml header search pa
+camlimages
Suspicious comments
Deprecated features
Fragile pattern matchings
Partially applied functions
Labels omitted
Overriden methods
Partial matches
Non unit statement
Edit
Based on:
None
Turn assertion checking off: assertions are not compiled.
[OCAML_NOASSERT, -noassert]


Target "Test depend" Info
General
Build
Rules
Comments
Configuration:
Active Configuration (Debug)
Collection:
OCaml linker
Setting
Value
Output type
Executable
Thread Model
No Threads
OCaml Libraries
str.cma
Custom Runtime
Link All
No autolink
Other OCaml linker flags
OCaml library search path
Edit
Based on:
None
This is a list included libraries (.cma) when linking Objective-Caml program. Libraries are delimited by whitespace. [OCAML_LIBRARIES]


Assistant
New Target
Special Targets
BSD
Carbon
Cocoa
Java
Kernel Extension
Legacy
OCaml
Shell Tool
Static Library
Cancel
Previous
Next