Photran 3.0 Developer's Guide

Jeffrey Overbey

Contents

Chapter 1

Introduction

Photran is a IDE for Fortran 90/95 and Fortran 77 that is built on top of Eclipse. It is structured as an Eclipse feature, in other words, as a set of plug-ins that are designed to be used together. Starting with version 3.0, it is an extension of CDT, the Eclipse IDE for C/C++. Previous versions of Photran were created by hacking a copy of the CDT to support Fortran instead of C/C++, but now we have developed a mechanism for adding new languages into the CDT, allowing the Fortran support code to be in its own set of plug-ins.

Our purpose in writing Photran was to create a refactoring tool for Fotran. Thus, Photran has a complete parser and program representation. Photran adds a Fortran editor and several preference pages to the CDT user interface, as well as a Fortran Managed Make project type.

This document explains the design of Photran so that you could fix a bug or add a refactoring. You should know how to use Photran and how the CDT works. You need to understand Eclipse and Eclipse plug-ins before you read this document. We recommend *The Java Developer's Guide to Eclipse.* for Eclipse newcomers.

1.1 CDT Terminology

The following are CDT terms that will be used extensively when discussing Photran.

- **Standard Make projects** are ordinary Eclipse projects, except that the CDT (and Photran) recognize them as being "their" type of project (as opposed to, say, projects for JDT, EMF, or another Eclipse-based tool). The user must supply their own Makefile, typically with targets "clean" and "all." CDT/Photran cleans and builds the project by running `make`.
- **Managed Make projects** are similar to standard make projects, except that CDT/Photran automatically generates a Makefile and edits the Makefile automatically when source files are added to or removed from the project. The **Managed Build System** is the part of CDT and Photran that handles all of this.
- **Binary parsers** are able to detect whether a file is a legal executable for a platform (and extract other information from it). The CDT provides binary parsers for Windows (PE), Linux (ELF), Mac OS X (Mach), and others. Photran does not provide any additional binary parsers.

- **Error parsers** are provided for many compilers. CDT provides a gcc error parser, for example. Photran provides error parsers for Lahey Fortran, F, g95, and others. Essentially, error parsers scan the output of `make` for error messages for their associated compiler. When they see an error message they can recognize, they extract the filename, line number, and error message, and use it to populate the Problems view.
- CDT keeps a **model** of all of the files in a project. The model is essentially a tree of **elements**, which all derive from a (CDT Core) class `ICElement`. It is described in the next section.

1.2 The Model

The Make Projects view in Photran is essentially a visualization of the CDT's *model*, a tree data structure describing the contents of all Make Projects in the workspace as well as the high-level contents (functions, aggregate types, etc.) of source files.

Alain Magloire (CDT) described the model, A.K.A. the `ICElement` hierarchy, in the thread "Patch to create ICoreModel interface" on the cdt-dev mailing list:

```
So I'll explain a little about the ICElement and what we get
out of it for C/C++.

The ICElement hierarchy can be separated in two:
(1) - how the Model views the world/resources (all classes above ITranslationUnit)
(2) - how the Model views the world/language (all classes below ITranslationUnit).

How we(C/C++) view the resources:
- ICModel  --> [root of the model]
    - ICProject --> [IProject with special attributes/natures]
        - ISourceRoot --> [Folder with a special attribute]
            - ITranslationUnit --> [IFile with special attributes, for example extensions *.c]
            - IBinary --> [IFile with special attributes, elf signature, coff etc...]
            - IArchive --> [IFile with special attributes, "<ar>" signature]
            - ICContainer -> [folder]

There are also some special helper classes
    - ILibraryReference [external files use in linking ex:libsocket.so, libm.a, ...]
    - IIncludeReference [external paths use in preprocessing i.e. /usr/include, ...]
    - IBinaryContainer [virtual containers regrouping all the binaries find in the project]

This model of the resources gives advantages:
- navigation of the binaries,
- navigation of the include files not part of the workspace (stdio.h, socket.h, etc ...)
- adding breakpoints
- search
- contribution on the objects
etc.....

[...]
```

```
(2) How we view the language.

Lets be clear this is only a simple/partial/incomplete view of the language.
For example, we do not drill down in blocks, there are no statements(if/else conditions) etc ....
For a complete interface/view of the language, clients should use the __AST__ interface.
```

From another one of Alain's posts in that thread:

```
Lets make sure we are on the same length about the ICElement hierarchy.
It was created for a few reasons:

- To provide a simpler layer to the AST.  The AST interface is too complex
  to handle in most UI tasks.
- To provide objects for UI contributions/actions.
- The glue for the Working copies in the Editor(CEditor), IWorkingCopy class
- The interface for changed events.
- ...

Basically it was created for the UI needs: Outliner, Object action contributions,
C/C++ Project view and more.

The CoreModel uses information taken from:
- the Binary Parser(Elf, Coff, ..)
- the source Parser(AST parser)
- the IPathEntry classes
- the workspace resource tree
- The ResolverModel (*.c, *.cc extensions), ...

to build the hierarchy.
```

1.3 The CDT Debugger and `gdb`

- The so-called CDT debugger is actually just a graphical interface to `gdb`, or more specifically to `gdb/mi`. If something doesn't work, try it in `gdb` directly, or using another `gdb`-based tool such as DDD.
- The debugger UI "contributes" breakpoint markers and actions to the editor. The "set breakpoint" action, and the breakpoint markers that appear in the ruler of the CDT (and Photran) editors are handled entirely by the debug UI: You will *not* find code for them in the Photran UI plug-in.
- `gdb` reads debug symbols from the executable it is debugging. That is how it knows what line it's on, what file to open, etc. Photran has *nothing* to do with this: These symbols are written entirely by the compiler. Moreover, the compiler determines what shows up in the Variables view. If the debugger views seem to be a mess, it is the compiler's fault, not Photran's.

Chapter 2

Plug-in Decomposition

Photran is actually a collection of several Eclipse plug-ins. The following projects comprise Photran as it is distributed to users.

- org.eclipse.photran.core

  This is the Photran Core plug-in. It contains all of the "behind the scenes" functionality that supports Fortran-specific things in the user interface:

  - IAdditionalLanguage for Fortran (i.e., the means of adding Fortran to the list of languages recognized by the CDT)
  - Error parsers for Fortran compilers
  - Fortran model builder and model elements
  - Interface to Fortran parser (stored in f95parser.jar)
  - Symbol table
  - Module database (and Berkeley DB Java Edition)
  - Refactorings, source printing, program manipulation
  - Non-UI (parsing and model building) preferences

- org.eclipse.photran.managedbuilder.core, org.eclipse.photran.managedbuilder.ui

  This is the Managed Build system. Craig Rasmussen (LANL) is handling it.

- org.eclipse.photran.modelicons

  This contains icons for all of the `FortranElement`s, i.e., the icons used in the Outline and Make Projects views, and provides a method which returns `ImageDescriptor`s for them.

- org.eclipse.photran.ui

  This contains the Fortran-specific components of the user interface:

  - Editor
  - Preference pages

- org.eclipse.photran-feature

  This is the Eclipse feature for Photran, used to build the Zip file distributed to users. (A feature is a grouping of related plug-ins; in our case, all of the plug-ins listed above.)

The following projects are in CVS but are not included directly in the Photran distribution:

- org.eclipse.photran-cdt-patches

  This contains CDT language extensibility/neutrality patches.

- org.eclipse.photran-dev-docs

  This contains developer documentation, including this manual.

- org.eclipse.photran-parser

  This contains the Fortran 95 parser, which is exported to f95parser.jar and stored in the Core plug-in. Note that this plug-in is not included directly in Photran and is not (should not be) referenced by any other plug-ins, since the parser is referenced through the JAR file (see parser description below).

- org.eclipse.photran.tests

  This contains JUnit tests for the parser, symbol table builder, source printer, program editor, refactorings, etc.

- org.eclipse.photran.xyzsamplelang

  This was my proof-of-concept for adding an additional language to the CDT. It is completely independent of Photran. It does not exist in the new Photran CVS repository and will soon be superceded by the Eightbol project.

Chapter 3

CDT Integration

Previous versions of Photran were developed by hacking the CDT: essentially, we made the user interface say "Fortran" instead of "C/C++," we replaced the model builder (which creates the Outline view, among other things) with one that ran a Fortran parser, we changed the list of file extensions, and we modified the syntax highlighting code to recognize Fortran comments and keywords.

Starting with Photran 3.0, we are taking a fundamentally different approach:

- We rename the C/C++ Perspective to the Make Perspective. We replace the four project types (C Standard Make Project, C++ Standard Make Project, C Managed Make Project, and C++ Managed Make Project) with two generic project types (Standard Make Project and Managed Make Project). We also change the launcher's menu item to read "Run Local Application" rather than "Run Local C Application." These are all superficial changes (i.e., changes to labels and icons, not to the underlying code). The Make project types use the natures formerly used for C projects.
- We add an extension point to the CDT which allows it to recognize other languages besides C and C++.

3.1 Integrating with the CDT: The `AdditionalLanguages` Extension Point

The org.eclipse.photran.xyzsamplelang project is a simple example of how to integrate a new language into the CDT. Its README is included in an appendix.

Essentially, the process of integrating a new language into the CDT works as follows:

- Create a new plug-in with an editor. (Obviously, you will eventually want to customize the editor for your language.)
- For each filename extension supported by your editor (i.e., each filename extension for files in your language), declare a (text) content type.

```
<extension point="org.eclipse.core.runtime.contentTypes">
```

```
   <content-type id="xyzSource" name="XYZ Language Source File"
         base-type="org.eclipse.core.runtime.text"
         priority="high"/>
   <file-association
         content-type="xyzSource"
         file-extensions="xyz"/>
</extension>
```

- Specify org.eclipse.cdt.core as a dependency of your plug-in, and declare in your `plugin.xml` that you are extending the org.eclipse.cdt.core.AdditionalLanguages extension point.

  ```
  <extension point="org.eclipse.cdt.core.AdditionalLanguages">
     <language class="addl_langs.XYZLanguage"/>
  </extension>
  ```

  You must provide the fully-qualified name of a class (`addl_langs.XYZLanguage`, in this case which implements the `org.eclipse.cdt.core.addl_langs.IAdditionalLanguage` interface.

- Fill in the definition of that class. See the JavaDoc in `IAdditionalLanguage` for more information.
  - `getName` just returns the name of the language (Fortran, XYZ Sample Language, etc.)
  - `getRegisteredContentTypeIds` returns a list of all of the content types you declared above. The content type names must be fully qualified, i.e., the name of the plug-in followed by the name of the content type, e.g., `XyzLanguagePlugIn.xyzSource` or `org.eclipse.photran.core.fortranSource`.
  - `createModelBuilder` returns a model builder for your language. Look at the `ToyModelBuilder` in the XYZ Sample Language project for a (trivial) example of how a model builder works, and see the JavaDoc for `IModelBuilder` for a more complete description. In essence,
    - It must implement `IModelBuilder`.
    - Its constructor takes a `TranslationUnit`, i.e., the file for which a model needs to be created.
    - The `parse` method return a Map taking `ICElements` to `CElementInfos`, which contains all of the elements that should appear in the Outline for the translation unit specified in the constructor. Use can either reuse the various implementations of `ICElement` in the CDT (`Macro, Namespace, FunctionDeclaration,` etc.), or you can create your own (with their own icons) by implementing `IAdditionalLanguageElement`.

- In the constructor for your editor, use the CDT UI plug-in's document provider rather than your own, i.e.,

  ```
  setDocumentProvider(CUIPlugin.getDefault().getDocumentProvider());
  ```

- Since you don't have your own document provider anymore, set up your partitioner by overriding

  ```
  protected void doSetInput(IEditorInput input)
  ```

  and setting it up there.

- Reuse the Outline page provided by CDT by copying the following field and methods from `FortranEditor` into your editor (notice that we are passing `null` into the constructor for `CEditorOutlinePage`; this doesn't seem to hurt anything):

```
protected CContentOutlinePage fOutlinePage;

public Object getAdapter(Class required) { ... }

public CContentOutlinePage getOutlinePage() { ... }

public static void setOutlinePageInput(CContentOutlinePage page,
        IEditorInput input) { ... }
```

- If you want your editor to jump to the correct location when something is selected in the Outline view, it needs to implement `ISelectionChangedListener`. (If you don't require this, remove the line "`fOutlinePage.addSelectionChangedListener(this);`" from the code copied above.) We implemented `ISelectionChangedListener` by copying the following methods verbatim from `CEditor`:

```
public void selectionChanged(SelectionChangedEvent event) { ... }

private boolean isActivePart() { ... }

public void setSelection(ISourceRange element, boolean moveCursor) { ... }
```

- If you want to be able to set breakpoints in your editor, add this line to the constructor for your editor class:

```
// JO: This gives you a "Toggle Breakpoint" action (and others)
// when you right-click the Fortran editor's ruler
setRulerContextMenuId("#CEditorRulerContext"); //$NON-NLS-1$
```

- If you want the CDT's Refactor menu to appear in your editor, add this line to the constructor for your editor class:

```
// JO: This will put the Refactor menu in the editor's
// context menu, among other things
setEditorContextMenuId("#CEditorContext"); //$NON-NLS-1$
```

3.2 Photran-CDT Integration

Photran is integrated into the CDT in the same way described above.

- The Photran Core plug-in defines a class `FortranLanguage`, which implements `IAdditionalLanguage`, and hooks it into the (new) `AdditionalLanguages` extension point in the CDT core.
- Photran's Core plugin.xml defines a content type `org.eclipse.photran.core.fortranSource`; `FortranLanguage` notifies the CDT that we want it to treat this content type as a valid source language.
- We provide a `FortranModelBuilder`, which runs our Fortran parser over a source file to create a CDT-compatible model.
  - Rather than using the CDT's model elements, we provide our own model elements, which are all subclasses of `FortranElement`. (`FortranElement` implements `IAdditionalLanguageElement`.)

  - The icons corresponding to `FortranElement`s are stored in the org.eclipse.photran.modelicons plug-in.[1]

- The Core plug-in also contributes several error parsers, which are described in the next section.

[1]The Photran UI plug-in would be the logical place to store these icons. However, the UI already has the Core as a dependency. The icon requests come from the `FortranElement`s in the Core, so if they had to be retrieved from the UI, we would introduce a circular dependency. So we put them in their own plug-in.

Chapter 4

The Photran Core Plug-in

The Photran Core plug-in (org.eclipse.photran.core) contains several source folders:

- `src` contains the main plug-in class (`FortranCorePlugin`), the Fortran implementation of `IAdditionalLanguage` (`FortranLanguage`), and any other "miscellaneous" classes that don't fit into one of the other folders.
- `errorparsers` contains a number of built-in error parsers for popular Fortran compilers.
- `model` contains the model builder for Fortran (`FortranModelBuilder`) and the Fortran model elements (`FortranElement` and nested subclasses).
- `parser` contains the `FortranProcessor` class, which is an interface to the parser, which is stored in `f95parser.jar`. It also contains all of the symbol table classes.
- `refactoring` contains everything related to refactoring that is not used elsewhere in Photran, for example, the `Program` and `Presentation` classes, the program editor, the source printer, etc. (All of these are described later.)
- `preferences` contains classes which "wrap" the various preferences that can be set in the Core plug-in. The actual preference pages displayed to the user are, of course, in the UI plug-in, but they use these classes to get and set the preference values.

The parser JAR (`f95parser.jar`) is contained in the root folder of this plug-in.

4.1 Error Parsers

Error parsers scan the output of `make` for error messages for a particular compiler. When they see an error message they can recognize, they extract the filename, line number, and error message, and use it to populate the Problems view.

For an example, see `IntelFortranErrorParser`. (It's a mere 74 lines.)

To create a new error parser,

- In package `org.eclipse.photran.internal.errorparsers`, define a class implementing `IErrorParser`
- Implement `public boolean processLine(String line, ErrorParserManager eoParser)` which should always returns false because ErrorParserManager appears not to use the result in a rational way
- In org.eclipse.photran.core's `plugin.xml`, find the place where we define all of the Fortran error parsers. Basically, copy an existing one. Your addition will look something like this:

```
<extension
      id="IntelFortranErrorParser"
      name="Photran Error Parser for Some New Fortran Compiler"
      point="org.eclipse.cdt.core.ErrorParser">
   <errorparser
         class="org.eclipse.photran.internal.errorparsers.MyErrorParser">
   </errorparser>
</extension>
```

- Your new error parser will appear in the error parser list in the Preferences automatically, and it will be automatically added to new projects. For existing projects, you will need to open the project properties dialog and add the new error parser to the project manually.

Note. Error parsers to not have to be implemented in the Photran Core plug-in. In fact, they do not have to be implemented in Photran at all. If you create a brand new plug-in, you can specify org.eclipse.cdt.core as a dependency, include the above XML snippet in your plug-in's `plugin.xml`, and include your custom error parser class in that plug-in.

Chapter 5

The Photran UI Plug-in

The Photran UI plug-in (org.eclipse.photran.ui) contains the Fortran editor and several preference pages.

Eclipse editors have a very non-intuitive structure which is very nicely explained elsewhere (for example, in *The Java Developer's Guide to Eclipse*).

5.1 Lexer-based Syntax Highlighting

The main difference between our Fortran editor and a "normal" Eclipse editor is that we do not use the typical means of syntax highlighting. Since Fortran does not have reserved words, keywords such as "if" and "do" can also be used as identifiers. So the word "if" may need to be highlighted as a keyword in one case and as a varible in another.

To do this, we actually run the Fortran lexical analyzer over the entire source file. It splits the input into tokens and specifies whether they are identifiers or not. We create a partition for each token. We also create a partition for the space between tokens. Each entire partition, then, is given a single color, based on its contents (keyword, identifier, or comments/whitespace). This is all done in the class `FortranPartitionScanner`.

Chapter 6

The Photran Lexers and Parser

The Fortran 95 lexers and parser are stored in the org.eclipse.photran.parser project. Because this project takes several minutes and a lot of memory (600-800 MB) to compile, it is exported as a (non-sealed) JAR archive, `f95parser.jar`, which is stored in the Core plug-in. The Core, Managed Builder, and UI use classes from this JAR file, not from the org.eclipse.photran.parser project. So you do *not* need the org.eclipse.photran.parser project in your workspace unless you will be regenerating this JAR file.

6.1 Accessing the Lexers and Parser from Photran: The `FortranProcessor`

(The following does *not* apply to JUnit tests.)

To scan or parse a file, or to build a symbol table from a parse tree, create a `FortranProcessor` and use it. It knows how to distinguish between fixed and free form file extensions based on the user's workspace preferences and can determine whether or not parser debugging has been enabled (also in the workspace preferences).

Examples:

- Parsing a file: See `FortranModelBuilder#parse`
- Lexing a file without parsing: See `FortranPartitionScanner#documentChanged`
- Creating a symbol table after a file has been parsed: (TODO-Jeff: No good examples yet)

6.2 Structure of the Lexers and Parser

Unfortunately, Fortran is not an easy language to process. It has two *source forms,* and keywords are not reserved (so it's fine to call a variable "if"). This greatly complicates lexical analysis. To make things as simple as possible, we have separated the lexical analysis into several phases, depending on whether the input is in free form or fixed form.

When a free form source file is being parsed...

- `FreeFormLexerPhase1`, which is generated from `FreeFormLexerPhase1.flex` by JFlex, splits the input into tokens. Tokens such as "if," "do," "len=," etc. are labeled as keywords, even if they are being used as identifiers (it doesn't know any better).
- `FreeFormLexerPhase2` reads the token stream from `FreeFormLexerPhase1` and buffers an entire statement worth of tokens. It then uses Sale's algorithm and an elaborate set of rules (specified in the method `buildAdditionalRules`) to determine which keyword tokens should be kept as keywords and which ones should actually be identifiers. In the case of "keywords" with an equal sign on the end (such as "len="), if that token should really be an identifier, it is split into two tokens: the identifier "len" and the = token T_EQUALS.
- The `Parser`, which is generated from `Fortran95.bnf` by my parser generator, reads the tokens from `FreeFormLexerPhase2` and parses the token stream, returning a parse tree. The parse tree is described in more detail later.

When a fixed form source file is being parsed...

- `FixedFormLexerPrepass` discards whitespace and comments and concatenates continuation lines.
- `FixedFormLexerPhase1`, which is generated from `FreeFormLexerPhase1.flex` by JFlex, splits the input into tokens. Essentially, it is identical to `FreeFormLexerPhase1`, except that there are no rules for whitespace, comments, or line continuations.
- `FreeFormLexerPhase2` reads the token stream from `FixedFormLexerPhase1`, resolving identifiers as it does for free form input.
- `FixedFormLexerPhase2` reads the token stream from `FreeFormLexerPhase2` and concatenates consecutive identifiers.
- The `Parser` reads the tokens from `FixedFormLexerPhase2`.

6.3 Parse Trees

6.3.1 Overview

NOTE. *We expect that you already know what a parse tree is and what the difference is between abstract and concrete syntax. Terms like "terminal," "nonterminal," and "token" should be familiar. If they are not, you will need to spend some time with the Dragon book*[1] *before tackling this section.*

When you call one of the `parse` methods on a `FortranProcessor`, either an exception will be thrown (if the lexer or parser encounters an error), or the method will return a (non-null) parse tree.

A parse tree is just a tree of `ParseTreeNode`s with `Token`s as leaves. The `ParseTreeNode` returned by the `parse` method is the root of the parse tree.

The parse tree for a program *exactly* follows the derivation of the program from the grammar in `Fortran95.bnf`. It is literally a parse tree, also known as a concrete syntax tree; it is *not* an abstract syntax tree (AST).[2]

[1] Aho, Sethi, and Ullman, *Compilers: Principles, Techniques, and Tools*

[2] If you want to know why we didn't use an abstract syntax tree, see the appendix on Miscellaneous Parser and Language Processing Notes.

It is important to remember that the grammar in `Fortran95.bnf` is different than the one in the Fortran standard. The grammar in the Fortran standard is *not* LALR(1)—not even close—and so it had to be modified (quite heavily) to successfully generate a parser from it. So the parse trees you get from our parser are *not* always going to match what the Fortran standard grammar would lead you to expect. For example, there is no SFExpr in the Fortran standard grammar, but you will find them popping up in your parse trees.

TIP. When running Photran, go into your workspace preferences; in the Fortran Parser section, there is an (very useful) option to "display parse tree rather than normal Outline view."

6.3.2 An Example

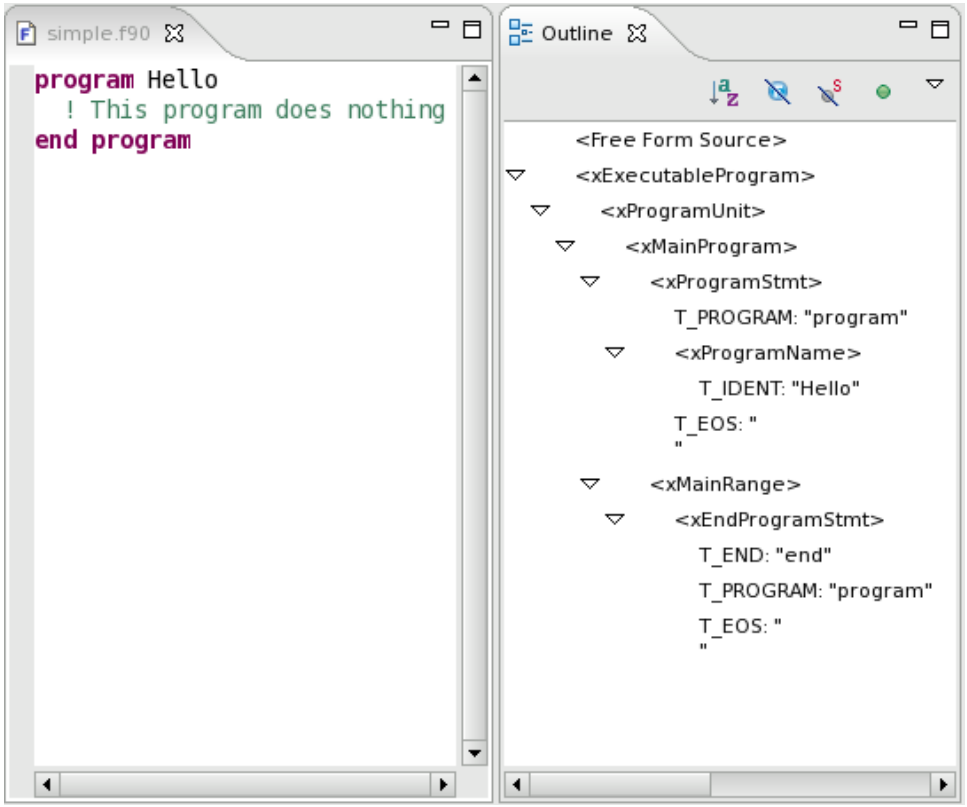

Figure 6.1: Sample program and its parse tree

Figure 6.1 shows a trivial program and its parse tree.

- The node labels in angle brackets, such as <xExecutableProgram> and <xProgramStmt>, are nonterminals in the grammar and correspond to `ParseTreeNodes` in the parse tree.[3]
- Terminals in the grammar have names starting with T_: for example, T_PROGRAM is the "program" keyword and T_IDENT is an identifier. Notice that the carriage return is also a token: end-of-statement (T_EOS). In the figure, these identify `Tokens` in the parse tree.[4]

[3]Note that <Free Form Source> (at the top of the Outline view) is just a label and not part of the parse tree.
[4]The distinction between terminals and tokens is somewhat subtle. Terminals are things like "identifier," "plus sign," and

Recall that each parse tree node corresponds to a nonterminal (on the left-hand side of a production in the grammar) and each token corresponds to a terminal.

You can determine the nonterminal corresponding to a `ParseTreeNode` by invoking `ParseTreeNode#getRootNonterminal`. (This method name will soon be changed to `getCorrespondingNonterminal` or something more intuitive like that.) This will return a `Nonterminal`. The only valid `Nonterminal`s are all constants in the `Nonterminal` class, so you can do identity comparisons like this:

```
if (parseTreeNode.getRootNonterminal() == Nonterminal.XEXECUTABLEPROGRAM) ...
```

Similarly, you can determine the terminal corresponding to a token by calling `Token#getTerminal` and doing an identity comparison, as in

```
if (token.getTerminal() == Terminal.T_IDENT) ...
```

You can get the text of the token ("Hello" for the only T_IDENT in Figure 6.1) by calling `Token#getText`.

`Terminal`s and `Nonterminal`s both have a `getDescription` method which returns a (String) description of that terminal or nonterminal, e.g., "T_IDENT" or "<xExecutableProgram>."

"float" that appear in a grammar. Tokens are the actual chunks of text from your input file, such as "hello," "+," and "3.14." Every token *corresponds* to a terminal, e.g., "hello" is an identifier, "+" is (the only representation of) a plus sign, and "3.14" is a float.

Chapter 7

Parse Tree Analysis: Visitors and Searching

NOTE. This section assumes that you are familiar with the Visitor pattern[1] *and are familiar with symbol tables*[2]*.*

7.1 Visitors

When you want to retrieve structural information about a program, typically it will be done using a Visitor on the parse tree.

There are two types of Visitors available in Photran.

- A `ParseTreeVisitor` has a "callback" method for each nonterminal in the grammar. The tree is traversed in preorder, and as each node is visited, a call is made to the Visitor method corresponding to the nonterminal for that node. This is usually the preferred Visitor to use.
- A `GenericParseTreeVisitor` only distinguishes between `ParseTreeNode`s and `Token`s. This Visitor should only be used when all internal nodes are treated similarly. If you need to distinguish between the types of internal nodes, use a `ParseTreeVisitor` instead.

(`ParseTreeVisitor` is generated by the parser generator and should not be modified.)

In addition to the visit methods, both types of Visitors have a `preparingToVisitChildrenOf` method, which is called after a node has been visited but before its children have, and a `doneVisitingChildrenOf` method, which is called immediately after its children have been visited but before any sibling nodes have.

[1] See Gamma, Helm, Johnson, and Vlissides, *Design Patterns*
[2] See Aho, Sethi, and Ullman, *Compilers: Principles, Techniques, and Tools*

7.1.1 Creating and Using a Visitor

Create a visitor by subclassing `ParseTreeVisitor` or `GenericParseTreeVisitor`. By default, all of the visit methods do nothing, so you only have to override the methods that are useful to you.

Given the root of the parse tree (i.e., the `ParseTreeNode` returned by `FortranProcessor#parse`), use the `visitUsing` method to traverse the parse tree; pass your visitor as the only argument.

7.1.2 Sample Traversals

As an example, consider again the "do nothing" program from Figure 6.1.

When it is visited using a `GenericParseTreeVisitor`, the Visitor methods are called in this order (methods called between `preparingToVisitChildrenOf` and `doneVisitingChildrenOf` are indented to illustrate that child nodes are being visited):

```
visitParseTreeNode(<xExecutableProgram>)
preparingToVisitChildrenOf(<xExecutableProgram>)
  visitParseTreeNode(<xProgramUnit>)
  preparingToVisitChildrenOf(<xProgramUnit>)
    visitParseTreeNode(<xMainProgram>)
    preparingToVisitChildrenOf(<xMainProgram>)
      visitParseTreeNode(<xProgramStmt>)
      preparingToVisitChildrenOf(<xProgramStmt>)
        visitToken(T_PROGRAM: "program")
        visitParseTreeNode(<xProgramName>)
        preparingToVisitChildrenOf(<xProgramName>)
          visitToken(T_IDENT: "Hello")
        doneVisitingChildrenOf(<xProgramName>)
        visitToken(T_EOS: (end of line))
      doneVisitingChildrenOf(<xProgramStmt>)
      visitParseTreeNode(<xMainRange>)
      preparingToVisitChildrenOf(<xMainRange>)
        visitParseTreeNode(<xEndProgramStmt>)
        preparingToVisitChildrenOf(<xEndProgramStmt>)
          visitToken(T_END: "end")
          visitToken(T_PROGRAM: "program")
          visitToken(T_EOS: (end of line))
        doneVisitingChildrenOf(<xEndProgramStmt>)
      doneVisitingChildrenOf(<xMainRange>)
    doneVisitingChildrenOf(<xMainProgram>)
  doneVisitingChildrenOf(<xProgramUnit>)
doneVisitingChildrenOf(<xExecutableProgram>)
```

When it is visited using a `ParseTreeVisitor`, the Visitor methods are called in this order:

```
visitXexecutableprogram(<xExecutableProgram>)
preparingToVisitChildrenOf(<xExecutableProgram>)
```

```
  visitXprogramunit(<xProgramUnit>)
  preparingToVisitChildrenOf(<xProgramUnit>)
    visitXmainprogram(<xMainProgram>)
    preparingToVisitChildrenOf(<xMainProgram>)
      visitXprogramstmt(<xProgramStmt>)
      preparingToVisitChildrenOf(<xProgramStmt>)
        visitXprogramname(<xProgramName>)
        preparingToVisitChildrenOf(<xProgramName>)
        doneVisitingChildrenOf(<xProgramName>)
      doneVisitingChildrenOf(<xProgramStmt>)
      visitXmainrange(<xMainRange>)
      preparingToVisitChildrenOf(<xMainRange>)
        visitXendprogramstmt(<xEndProgramStmt>)
        preparingToVisitChildrenOf(<xEndProgramStmt>)
        doneVisitingChildrenOf(<xEndProgramStmt>)
      doneVisitingChildrenOf(<xMainRange>)
    doneVisitingChildrenOf(<xMainProgram>)
  doneVisitingChildrenOf(<xProgramUnit>)
doneVisitingChildrenOf(<xExecutableProgram>)
```

(The Visitors used to generate the output above are stored in a JUnit test class called `ExampleVisitor`.)

7.2 The `ParseTreeSearcher`

Inside Visitor methods, typically you will want to look for particular tokens or syntactic structures.

7.2.1 An Example

Consider the following (randomly selected) method from `DeclarationCollector`, one of the classes used for building symbol tables. This class scans a program's parse tree, looking for declarations of programs, functions, block data, namelists, etc., and inserts declarations for them into a symbol table.

```
    public void visitXmainprogram(ParseTreeNode node)
    {
        // # R1101 desn"t ensure ordering as the standard requires;
        // <xMainProgram> ::=
        //   <xMainRange> |
        //   <xProgramStmt> <xMainRange> ;
        // # R1102
        // <xProgramStmt> ::=
        //   <xLblDef> T_PROGRAM <xProgramName> T_EOS ;
        ParseTreeNode programStmt = ParseTreeSearcher.findFirstNodeIn(node, Nonterminal.XPROGRAMSTMT);
        Token name;
        if (programStmt != null)
        name = ParseTreeSearcher.findFirstIdentifierIn(node);
        else
        {
```

```
        name = new Token();
        name.setText("(Anonymous Main Program)");
        }
        ...
```

(The comment lines are copied from the parser generation grammar, `Fortran95.bnf`.)

In Fortran, a main program can either start with a program statement, or it can not. So both of these are valid programs:

```
program SayHi
  print *, 'Hello!'
end program SayHi
```

```
  print *, 'Hello!'
end program SayHi
```

In this Visitor method, `node` is guaranteed to be a <xMainProgram>, due to the name of the method. As shown, by looking at the grammar, we can determine that it will either have one child (a `ParseTreeNode` corresponding to an <xMainRange>) or two children (both `ParseTreeNode`s, the first corresponding to an <xProgramStmt> and the second corresponding to an <xMainRange>.

If it *does* have an <xProgramStmt>, then, a quick look at the rules for <xLblDef> (an optional integer label that can appear at the beginning of any Fortran statement) and <xProgramName> will make it evident that the first identifier token (T_IDENT) under the <xProgramStmt> is the name of the program.

So now we can understand what this method does.

- It checks to see if the main program has a <xProgramStmt>. This is done by calling `ParseTreeSearcher#findFirstNod` and telling it we want to find the first <xProgramStmt> that is a child of `node`. It will either return a `ParseTreeNode` corresponding to an <xProgramStmt>, or if it can't find one, it will return `null`.
- If there is an <xProgramStmt>, it grabs the first T_IDENT token, which tells the name the user gave to the program.
- If there was no <xProgramStmt>, the program does not have a name, so we "fake" a token and name the program "(Anonymous Main Program)."

7.2.2 For More Information

So, essentially, anytime you have a parse tree (or a subtree of the parse tree) and you want to find a particular node or token, use one of the methods in `ParseTreeSearcher`. If there isn't one, you may have to write one yourself. Be sure to look at the JavaDoc for the methods in that class; unless you are doing something bizarre, one of the existing methods should work.

Chapter 8

Symbol Tables

All of the symbol table routines are stored in the "parser" folder under the Core plug-in.

After you have parsed a file, you can create a symbol table for it by calling `FortranProcessor#createSymbolTableFromParse`
This, in turn calls the static method `SymbolTable#createSymbolTableFor`, which is intended to be the sole
entrypoint for symbol table building.

8.1 Contents of a Symbol Table

The classes representing symbol table entries are stored in the `org.eclipse.photran.internal.core.f95parser.symboltab`
package. Currently, the following are allowed.

- Main Programs
- Modules
- Functions
- Subroutines
- Derived Types
- Block Data
- Namelists
- Common Blocks
- Interfaces
- Externals
- Intrinsics
- Variables

`FortranProcessor#createSymbolTableFromParseTree` returns a `SymbolTable`, which represents the global symbol table for the program that was parsed.

A `SymbolTable` is essentially just a collection of `SymbolTableEntry` objects. Each `SymbolTableEntry` has a child symbol table. For `FunctionEntry` objects, this child table contains all of the parameters, the return variable, and any local variables declared inside the function. For `VariableEntry` objects, which represent local variables, the child table will always be empty (it is there only for uniformity).

Symbol tables also keep track of whether an "implicit" rule applies, what external modules are being used (via USE statements), etc.

To see what's in a symbol table, just use the `toString` method. Child tables will be indented in the output.

The symbol table-related classes are JavaDoc'd, so additional information is available there.

8.2 `SymbolTableVisitors`

You can also create a Visitor for a program's symbol table hierarchy by subclassing `SymbolTableVisitor`, which has a visit method for each type of `SymbolTableEntry`.

8.3 The Module Database

Similar to *import* statements in Java, Fortran programs can USE a module defined in a different Fortran file. Unfortunately, there is no easy way to tell where this module might be defined. The user simply specifies a list of "module paths" which are searched for Fortran source files. Each Fortran source file must be checked to see if it contains a module with the given name.

In Photran, the list of modules paths is stored in a workspace preference, although we plan to convert this to a project property.

The class `ModuleLoader` is responsible for locating modules in this way. The `ModuleDB` caches the symbol tables for files on the module path so they don't all have to be reparsed every time a module is searched for. Neither of these is complete yet, but they will be soon.

8.4 How Symbol Tables are Built

A quick look at `FortranProcessor#createSymbolTableFromParseTree` explains how symbol tables are built:

```
public static SymbolTable createSymbolTableFor(ParseTreeNode parseTree) throws Exception
{
    SymbolTable tbl = (new DeclarationCollector(parseTree)).getSymbolTable();
    Intrinsics.fill(tbl);
    return (new ReferenceCollector(tbl, parseTree)).getSymbolTable();
}
```

- A Visitor is run over the parse tree to collect declarations of programs, modules, functions, subroutines, block data, derived types, namelists, etc.—anything that can be stored in the symbol table.
- The names of all Fortran 95 intrinsic functions and subroutines are inserted into the table.
- Now that declarations have been inserted, the parse tree is scanned for references (i.e., uses of a function, variable, namelist, etc.). If implicit variables have been allowed, the `ReferenceCollector` will detect those (since they are used but not declared) and insert them into the symbol table.

Chapter 9

Refactoring Support: Source Printing and Program Editing

Aside from the usual front end components (parser and symbol tables), a refactoring tool requires

- a means of manipulating the parse tree, i.e., moving nodes around, deleting them, and inserting new ones; and
- a means of outputting "revised" source code after a parse tree has been manipulated.

We will look at the means of outputting source code first, discussing the `Presentation` and `SourcePrinter` classes. We will then discuss the `ProgramEditor`, which allows you to manipulate the parse tree and `Presentation`.

9.1 The `Presentation` and the `Program`

While the parse tree for a program stores all of the "important" tokens (identifiers, parentheses, etc.), other things—comments, line continuations, and C preprocessor directives—are not in the Fortran grammar. However, when the source printer produces source code from a parse tree, it needs to include these as well.

We refer to these things (comments, line continuations, C preprocessor directives, Fortran INCLUDE statements, and anything else that does not end up in a parse tree) as *non-tree tokens*, and we represent them by the class `NonTreeToken`. A `Presentation` is essentially a list of all the non-tree tokens that appeared in a parse.

A `Presentation` can be created from a parse by calling the `getNonTreeTokens` method on the lexer and passing the resulting `List<NonTreeToken>` to the `Presentation` constructor.

A `Program` is just parse tree paired with a symbol table and a `Presentation`.

9.2 Presentation Blocks

Since `Token`s in the parse tree and `NonTreeToken`s in a program's `Presentation` have a lot in common, we will refer to them collectively as "presentation blocks." Not surprisingly, they share a common superclass (interface, rather): `IPresentationBlock`. JavaDoc removed, it looks like this:

```
public interface IPresentationBlock
{
    public abstract String getFilename();
    public abstract void setFilename(String value);

    public abstract int getStartLine();
    public abstract void setStartLine(int value);

    public abstract int getStartCol();
    public abstract void setStartCol(int value);

    public abstract int getEndLine();
    public abstract void setEndLine(int value);

    public abstract int getEndCol();
    public abstract void setEndCol(int value);

    public abstract int getOffset();
    public abstract void setOffset(int value);

    public abstract int getLength();
    public abstract void setLength(int value);

    public abstract String getText();
    public abstract void setText(String value);
}
```

Intuitively, then, all presentation blocks know what file they originated from, where they were in the file (both in terms of line/column and offset/length), and exactly what their text looked like. (This is important since capitalization does not matter in Fortran.)

Most importantly, this means that reproduce the source code of a program verbatim from a parse tree and a `Presentation`. (The only difference will be the use of spaces vs. tabs to make sure tokens appear in the correct column on a line.) All comments and formatting will be retained.

9.3 The `SourcePrinter`

The job of the source printer (class `SourcePrinter`) is to take a a parse tree and corresponding `Presentation` and produce source code from it.

Since every `Token` in the parse tree and every `NonTreeToken` in the `Presentation` knows what line and column it should appear on, this is easy.

9.4 The `ProgramEditor`

Essentially, a refactoring needs to change the parse tree for a program. It may change existing nodes, move them around, remove them altogether, or insert new nodes.

As described above, though, source code is produced by looking at the line/column offsets of the `Token`s in the parse tree and interleaving comments and other non-tree tokens from a program's `Presentation`.

When we add, move, change, or delete a subtree of the parse tree, then, we must do three things:

- adjust the positions of the `Token`s in that subtree,
- adjust the positions of the related `NonTreeToken`s (e.g., the comments describing a method and C preprocessor directives in its definition)
- adjust the positions of all of the presentation blocks that appear after the modified subtree. For example, if you change an token's text from "Hello" to "Goodbye," every presentation block after that one will have its offset incremented by 2, and every presentation block to the right of that token on the same line will also have its starting column number incremented by 2.

The (static) methods in `ProgramEditor` are used to add, move, modify, and delete subtrees.

This class will be described more as it stabilizes.

Chapter 10

Refactoring: Preconditions and Implementation

From an implementation standpoint, a refactoring consists of

- a set of *initial preconditions*,
- input from the user,
- a set of *final preconditions*, and
- a program manipulation.

As an example, consider the easiest of all refactorings: Rename.

- **Initial preconditions:**
  - The token to rename should be an identifier.
  - The identifier must correspond to an entry in the symbol table.
  - If Rename is limited to certain entities, say, variables and subprograms, the symbol table entry should indicate that the identifier corresponds to a variable or subprogram.
- **User input:**
  - New name for the variable or subprogram
- **Final preconditions:**
  - The new name must be a valid identifier.
  - The new name must be different from the old name.
  - The new name must not already exist in its namespace.
  - For every reference to the old name, a change to the new name should uniquely identify the same variable or function. For example, if you are renaming a global variable from `a` to `b`, but a local variable `b` will end up shadowing the global variable and cause one of its references to "go wrong," then the rename cannot continue.

- **Program manipulation:**
  - Locate the declaration of the entity being renamed, and locate all references in all files in the workspace, and change the token text to the new name.

The distinction between initial and final preconditions, then, is that initial preconditions must be satisfied before the user is asked for input, while final preconditions depend on the user's input. If a refactoring does not require user input, it will have no final preconditions.

10.1 Running a Refactoring

Running a refactoring looks something like this.[1]

```
RenameRefactoring r = new RenameRefactoring(getProgram(), targetToken);
if (!r.checkInitialPreconditions()) throw new Error(r.getErrorMessage());
r.setNewName("Whatever");
if (!r.checkFinalPreconditions()) throw new Error(r.getErrorMessage());
if (!r.perform()) throw new Error(r.getErrorMessage());
```

In other words, you

- check the initial preconditions,
- get input from the user,
- check the final preconditions, and
- finally perform the refactoring.

At any point, if a step has failed, you can call `r.getErrorMessage()` to get an explanation suitable for displaying to the user.

10.2 The `FortranRefactoring` Class

All Fortran refactorings must be subclasses of `FortranRefactoring`. `FortranRefactoring` (or its superclass) provides

- an instance variable `error`, the contents of which will be returned when the user calls `getErrorMessage()`. If the refactoring fails, before returning false, be sure to set this so the user knows why.
- Two `List`s of `Precondition`s: `initialPreconditions` and `finalPreconditions`. Add preconditions to the former in the constructor and the latter after input has been received from the user.
- Two fields, `initialPreconditionsCheckedAndPassed` and `finalPreconditionsCheckedAndPassed`. For example, you will want to assert that the initial preconditions have been checked and passed before checking the final preconditions.

[1] The `if (!...) throw ...` is an obvious code smell that makes it look like the various methods in `RenameRefactoring` should be throwing exceptions rather than returning booleans. The current structure makes more sense in "real" code, where the user is being assaulted with dialog boxes and other things happen between each of the steps.

10.3 Preconditions

Refactoring preconditions are stored in the package `org.eclipse.photran.internal.core.refactoring.preconditions`. They are all derived from the class `AbstractPrecondition`.

A precondition (i.e., a class derived from `AbstractPrecondition`) has:

- a `List` of prerequisite preconditions, and
- a method for checking this precondition.

To check a precondition, call its `check` method. After this method has been called once, it "remembers" whether it succeeded or failed, so future calls to `check` will just return the stored result rather than performing the entire check again.[2]

To implement how a precondition is checked, override the abstract method `checkThisPrecondition` method. If any other preconditions need to be checked before this one, add them to the `prereqPreconditions` list in the constructor. If the code in `checkThisPrecondition` is called, they have all been satisfied.

The fields `parseTree`, `presentation`, and `symbolTable` will be populated when the constructor is called. You will almost definitely need to use these in your implementation of `checkThisPrecondition`.

10.4 Implementing a Refactoring

- If you need any preconditions that don't exist yet, implement them as described above.
- Create a subclass of `FortranRefactoring`.
- In the constructor, call `super` and add preconditions to the `initialPreconditions` field.
- Implement any methods to store input from the user. At the beginning of these methods, you will probably want to assert `initialPreconditionsCheckedAndPassed`.
- Implement `perform`. You will want to assert that all user input is in place as well as asserting `finalPreconditionsCheckedAndPassed`. Use the `ProgramEditor` to modify the parse tree and presentation. If your changes might possibly affect the program's symbol table, call the (inherited) `rebuildSymbolTable` method after all transformations have been completed.

TODO-Jeff: Figure out and document how to integrate a refactoring into the (CDT) UI.

[2] It is very possible that a precondition will be checked manually, and then it will be checked again because it is a prerequisite for another precondition. This resolves any inefficiences that might result because of this.

Appendix A

Getting the Photran 3.0 Sources from CVS

Part I. Check out the CDT sources from CVS

1. In Eclipse, switch to the CVS Repository Exploring perspective.
2. Right-click the CVS Repositories view; choose New, Repository Location
3. Enter the following information, then click Finish:

Host name:	dev.eclipse.org
Repository path:	/cvsroot/tools
Connection type:	pserver
Username:	anonymous
Password:	(no password)

4. Right-click on :pserver:anonymous@dev.eclipse.org:/cvsroot/tools, and choose Refresh Branches...
5. Select the following, and click OK:
   - org.eclipse.cdt-build
   - org.eclipse.cdt-core
   - org.eclipse.cdt-doc
   - org.eclipse.cdt-debug
   - org.eclipse.cdt-launch

   When prompted, tell it to Search Deeply.
6. Expand :pserver:anonymous@dev.eclipse.org:/cvsroot/tools, and then expand Versions (in the CVS Repositories view)
7. Under org.eclipse.cdt-build, expand org.eclipse.cdt-build CDT_3_0
8. Right click and check out all of the org.eclipse.cdt.* packages EXCEPT for the ones ending in "tests" (why bother testing?)

9. Do the same with org.eclipse.cdt-core, org.eclipse.cdt-debug, org.eclipse.cdt-doc, and org.eclipse.cdt-launch

10. You now have the CDT source code. Make sure it compiles successfully (lots of warnings, but no errors).

Part II. Check out the Photran source and the CDT patches

11. In Eclipse, switch to the CVS Repository Exploring perspective.

12. Right-click the CVS Repositories view; choose New, Repository Location

13. Enter the following information, then click Finish:

	Host name:	dev.eclipse.org
	Repository path:	/cvsroot/technology
	Connection type:	extssh
	Username/passwd:	(your eclipse.org committer username and password)
If you are a Photran committer:	*Otherwise:* Host name:	dev.eclipse.org
	Repository path:	/cvsroot/technology
	Connection type:	pserver
	Username:	anonymous
	Password:	(no password)

14. Expand the node for dev.eclipse.org:/home/technology, then expand HEAD (in the CVS Repositories view), then expand org.eclipse.photran

15. Right-click and check out all of the projects under org.eclipse.photran EXCEPT org.eclipse.photran-parser. You only need this project if you will be regenerating the parser from the grammar. (The parser is included in the org.eclipse.photran.core plug-in as a JAR file. This way, the parser does not have to be recompiled every time you rebuild Photran.)

    The sources will NOT compile at this point; you must complete the following...

Part III. Patch the CDT sources

16. Go to a bash prompt. Change to your Eclipse workspace directory (the one containing all of the org.eclipse.cdt and org.eclipse.photran projects).

17. `cd org.eclipse.photran-misc/photran30-cdt30-patches`

18. Run ./install

19. Go back into Eclipse. Refresh all of the org.eclipse.cdt packages. (Click the first, shift-click the last, right-click, choose Refresh.)

    The sources should all compile (albeit with about 640 warnings).

Note. *Committers working on the parser or refactoring engine should have access to /usr/local/cvsroot on brain.*

Appendix B

XYZ Language Plug-in README

The org.eclipse.photran.xyzsamplelang project demonstrates how to use the `AdditionalLanguages` extension point. It is just the sample XML editor plug-in that gets created when you choose to create a new plug-in project and select the "plug-in with an editor" template. However, it has been modified to integrate with the CDT. Here is the README from that project, which is a cursory description of how the CDT was modified and how the XYZ language was integrated. The specific changes to the editor are documented in its code, so be sure to read that too!

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *


   HOW TO TIE A NEW SOURCE FILE TYPE, PARSER, AND MODEL BUILDER INTO THE CDT
                    (AND HOW TO MAKE THE CDT ALLOW IT)


* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

If you create your own editor, model builder, and ICElement-derived elements,
some simple changes to the CDT source code will make the CDT
integrate your parser and the elements it produces into its model.

We add an AdditionalLanguages extension point to the CDT Core and change a
couple of methods to make use of it.

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

NOTE ON INCLUDED SOURCE CODE:

All files in the CDT Integration Proof-of-Concept - Phase 1 source folder were
generated by the New Plug-In wizard EXCEPT:

- Several changes were made to the editor's main class (XyzLanguageEditor) and
  are commented there
- I added FortranContentOutlinePage
- I added the ToyElement and ToyModelBuilder classes (based on Photran's
```

```
  FortranElement hierarchy and its (hidden) ToyModelBuilder)
- I added XyzLanguagePerspective, which gives us our own perspective and adds
  a shortcut for our new file wizard to the New File menu

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

This is NOT well-written, and it is NOT a tutorial.  It assumes you have some
idea of how the CDT works (e.g., what the Model is), and it assumes that you
will look at my code to see all the details.

The XYZ Sample Language code is documented, so that should be read in addition
to this README.

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

First, I checked out the CDT source (most of it, anyway) from CVS.

I created a basic editor and a New wizard to complement it
using the New Plugin wizard.  I just called it XyzLanguageEditor
rather than XMLEditor or whatever the default is.  The filename extension
is .xyz.

I added the CDT Core and UI plugins as dependencies of my new plugin.

After the wizard finished, I declared an xyzSource content type (text) to
match the .xyz filename extension that the editor uses.

(I also created an XYZ Language perspective by subclassing CPerspective, although
that has only cosmetic value and isn't necessary for what I describe below.)

-----

Before we can add additional languages to the CDT, we must make the following
changes to the CDT itself:

Add an AdditionalLanguages extension point to the Core's plugin.xml
Add AdditionalLanguages.exsd to the Core's schema folder
Add org.eclipse.cdt.core.addl_langs package to the Core's src folder, containing:
IAdditionalLanguage -- Each extension language must implement this
AdditionalLanguagesExtension.java -- Provides access to extension point; iterable
AdditionalLanguagesIterator.java -- Iterates through contributed IAdditionalLanguages
IAdditionalLanguageCallback -- For performing an action on each contributed language
IModelBuilder -- Extension languages provide their model builder this way
IAdditionalLanguageElement -- ICElement extensions must implement this

Now we need to make the CDT recognize our additional content types as
valid TranslationUnits in its model.

These changes make sure CoreModel#getRegistedContentTypeId works for
additional content types.  This function is called by CContainer
and, if it returns a valid (registerd-with-the-CDT) content type,
```

makes a TranslationUnit out of the file being processed.

Essentially, we just add a line or two to each of the following which asks the
AdditionalLanguagesExtension to check whether some extension plug-in (like our
XYZ Language Plug-in) supports a given content type.

1. CoreModel#getRegistedContentTypeIds
2. CCorePlugin#getContentType
3. TranslationUnit#isSourceUnit
4. CoreModel#isValidSourceUnitName
5. CoreModel#isValidTranslationUnitName

NOTE:
The following are places where the CCorePlugin.CONTENT_TYPE_CSOURCE
content type is checked for but I elected not to check for
extension content types:
TranslationUnit#isCLanguage
CoreModel#isValidCSourceUnitName
AbstractIndexerRunner#shouldIndex
...the following two methods are identical...
DOMSearchUtil#getLanguage
InternalASTServiceProvider#getLanguage

Next, we want to allow additional languages to be parsed by their own parser,
and they should be able to build their own models for the Outline and Make
Projects views.

The CModelBuilder is called by TranslationUnit#parse. We will make
TranslationUnit#parse call our own model builder (ToyModelBuilder), which
will use some special ICElements (base class ToyElement) to extend the C Model.
Again, this is done through the extension point, so it can apply to any language.

- CModelBuilder changed to implement IModelBuilder
- Added extension point checking to TranslationUnit#parse
- Made CElementInfo public (rather than default)
- Made CElementInfo#setIsStructureKnown public (rather than protected)
- Added extension point checking to CElementImageProvider#getBaseImageDescriptor

The last point is important. Additional languages can reuse the CDT's model elements
(for functions, classes, etc. -- all the things that show up in the Outline), or
they can create their own (e.g., Fortran has Modules and Block Data, neither of
which have direct analogs in C/C++). These new elements can be created by implementing
IAdditionalLanguageElement. IAdditionalLanguageElements must implement a
method getBaseImageDescriptor() which provides an Outline icon for that element.

To integrate our XYZ language into the CDT...

First, we add the following to our plugin.xml:

```
   <extension point="org.eclipse.cdt.core.AdditionalLanguages">
      <language class="addl_langs.XYZLanguage"/>
   </extension>
```

We provide a class addl_langs.XYZLanguage which implements IAdditionalLanguage.
See the JavaDoc for IAdditionalLanguage. Essentially, we claim to support
the XyzLanguagePlugIn.xyzSource content type, and we provide a ToyModelBuilder
which will provide a (static, and very boring) Outline of XYZ Language source files.

Next, I added Outline support to the editor by making a tiny subclass of
CContentOutlinePage. See several relevant notes in XyzLanguageEditor.java.
See also FortranContentOutlinePage. We are telling CContentOutlinePage
its editor is null, since it doesn't do anything useful with it.

Appendix C

Overview of Implementation of the CDT `AdditionalLanguages` Extension Point

The following, from an e-mail to the CDT copied to photran-dev, is an alternative description of how we modified the CDT to include this extension point.

```
1. Add an AdditionalLanguages extension point to the Core's plugin.xml and AdditionalLanguages.exsd to t

   Extend via <language class="my.plugin.XyzLanguage">
   where XyzLanguage implements IAdditionalLanguage (see below)

2. Add a package org.eclipse.cdt.core.addl_langs containing:

   IAdditionalLanguage
        public interface IAdditionalLanguage {
          public String getName();
          public boolean matchesSourceContentType(String contentTypeID);
          public Collection/*<String>*/ getRegistedContentTypeIds();
          public IModelBuilder createModelBuilder(TranslationUnit tu,
              Map newElements);
        }

   AdditionalLanguagesExtension.java
        Singleton; provides access to the extension point
        Methods:
            public Iterator/*<IAdditionalLanguage>*/ iterator()
            public void processAdditionalLanguages(
                IAdditionalLanguageCallback callback)
            public boolean someAdditionalLanguageMatchesContentType(
                String contentTypeID)
            public IAdditionalLanguage getLanguageForContentType(
                String contentTypeID)
```

```
   AdditionalLanguagesIterator.java -- see iterator() above
        Implements Iterable/*<IAdditionalLanguage>*/

   IAdditionalLanguageCallback -- see processAdditionalLanguages() above
        Allows you to perform some arbitrary action on each contributed
            IAdditionalLanguage

   IModelBuilder
        Each extension language provides a model builder this way
        Single method:
            public abstract Map parse(boolean quickParseMode)
                throws Exception;

   IAdditionalLanguageElement (extends ICElement)
        Allows you to extend the ICElement hierarchy
        Methods:
            public abstract Object getBaseImageDescriptor();
            - The return type should really be ImageDescriptor,
              but I don't want to make the Core depend on JFace

3. Change content type checking to use extension point...
   i.   CoreModel#getRegistedContentTypeIds
   ii.  CCorePlugin#getContentType
   iii. TranslationUnit#isSourceUnit
   iv.  CoreModel#isValidSourceUnitName
   v.   CoreModel#isValidTranslationUnitName

   The change each of these is just a line or two -- usually a call to
   AdditionalLanguagesExtension#someAdditionalLanguageMatchesContentType

4. Make CModelBuilder implement IModelBuilder (no substantive change)

5. Change the beginning of TranslationUnit#parse(Map):
        IModelBuilder modelBuilder;
        IAdditionalLanguage lang = AdditionalLanguagesExtension
            .getInstance()
            .getLanguageForSourceContentType(fContentTypeID);
        if (lang != null)
            modelBuilder = lang.createModelBuilder(this, newElements);
        else
            modelBuilder = new CModelBuilder(this, newElements);

6. Make CElementInfo public (rather than default)

7. Make CElementInfo#setIsStructureKnown public (rather than protected)

8. Add this to the top of CElementImageProvider#getBaseImageDescriptor:
   if (celement instanceof IAdditionalLanguageElement)
     return (ImageDescriptor)
       ((IAdditionalLanguageElement)celement).getBaseImageDescriptor();
```

Appendix D

Miscellaneous Parser and Language Processing Notes

D.1 Notes on the (almost-)LALR(1) Grammar

The current grammar is based on an Eli grammar that has been worked on for several years. All of the lexer work is totally original, and we fixed several bugs in the Eli grammar, but for the most part the grammar in `Fortran95.bnf` is the work of other people. We have invested several months of work in it—but that does not compare to the years of work invested by the previous authors of that grammar.

More information on the Eli grammar is available at http://members.aol.com/wclodius/eli_f95_parser.html

D.2 Why not an abstract syntax tree?

We started by building an AST for Fortran by hand. For a compiler, this wouldn't be a big deal. The purpose of an AST is provide a tree that only contains "useful" information from the parse, so superfluous tokens like parentheses never end up in an AST. For a refactoring tool, though, it is important to remember every token in the parse, because you need to reproduce code that looks similar to the user's original code. Fortran has a number of cases where there are several different ways to specify the same thing. For example, you can end a program with "end," "end program," or "end program" followed by the name of the program. Other cases, with several optional tokens in the middle of statements, are trickier. So, long story short, after a couple of months, we had about 600 AST classes and were nowhere near finished.

So we needed a different alternative.

One would be to have the parser generator generate the AST stuff for us. However, aside from the fact that it would require lots of tedious annotations in the grammar, we would still be in a place of having several hundred AST classes and a Visitor with several hundred methods.

Instead, we[1] decided to use a good old-fashioned parse tree.

[1] Actually, "I" is more correct... Spiros was leaving for Google, Brian was indifferent, and I think Dr. Johnson still would have preferred an AST... so now you know where to put the blame...

Here's why.

First, it made the "superfluous token" problem go away. Since each node just had a `List` of child nodes (either tokens or other parse tree nodes), we did not have to do anything extra to accommodate all of the variations of a statement. All the tokens were there when we needed them (e.g., for prettyprinting), and they could be ignored when we didn't need them.

Second, it gave us two possible Visitors, as described above. More importantly, unlike visiting an AST, these Visitors could just do a "dumb" recursive traversal of the tree, totally ignorant of the underlying grammar. This also made parse tree searches and grammar modifications easier.

There are probably other reasons as well which I can try to remember if someone is still not convinced that this was a good idea.