Parsing and Analyzing C/C++ code in Eclipse

Mike Kucera
IBM Toronto Software Lab

McMaster University
Hamilton, Ontario, Canada
4TB3/6TB3
March 1, 2012

© 2012 IBM Corporation; made available under the EPL v1.0

eclipse

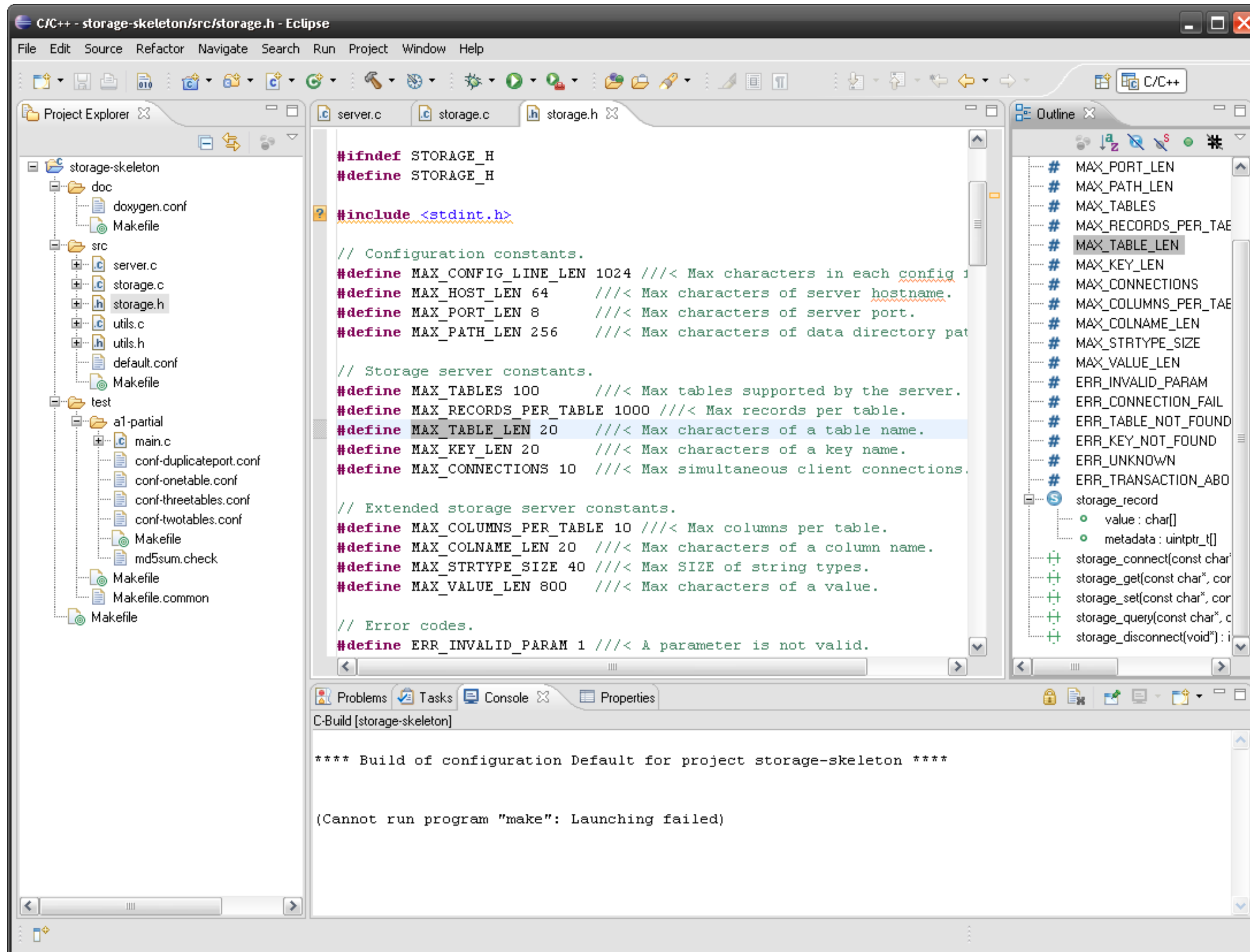
- IDE – Integrated Development Environment

- Supports several programming languages and paradigms
  - C/C++, Java, Scala, PHP, Ruby, COBOL, XML, HTML, etc…
  - Very popular as a Java IDE
- Multi-platform
  - Runs on Windows, Unix, Mac…
- Its open source
- Its free!

- Eclipse CDT project
  - Set of plug-ins that adds full support for developing C/C++ applications

© 2012 IBM Corporation; made available under the EPL v1.0

CDT Editor - DEMO



© 2012 IBM Corporation; made available under the EPL v1.0

CDT Index

- CDT parses and analyzes your code
  - Not just a text editor, eclipse “understands” your code
- CDT “compiles” the code into an index file
  - Designed for fast queries and searches

- Example: invoking “open declaration” on a function call will query the index to find the location of the declaration

- Index is built when you first create a project (assuming you have some existing code)
- Index is incrementally updated every time a file is changed

© 2012 IBM Corporation; made available under the EPL v1.0

CDT Index

- Index stores information about:
  - Identifiers and how they relate to each other
    - Called bindings
  - The locations (source file and offset) of each identifer
  - All the macros defined in each file
  - The include relationship between files
  - TODO comments

© 2012 IBM Corporation; made available under the EPL v1.0

eclipse

CDT Core

- Preprocessor
  - Converts text into a token stream, evaluates #directives and macros
- Parsers (C and C++)
  - Converts the token stream in to an AST
- AST
  - Visitor API
- AST Rewrite API
  - Used to implement refactoring
- Semantic analysis (name resolution)
  - Resolves the relationships between identifiers
- Indexer
  - Generates and updates the index file by processing the AST
- Index API
  - Allows index based tooling to query the index

© 2012 IBM Corporation; made available under the EPL v1.0

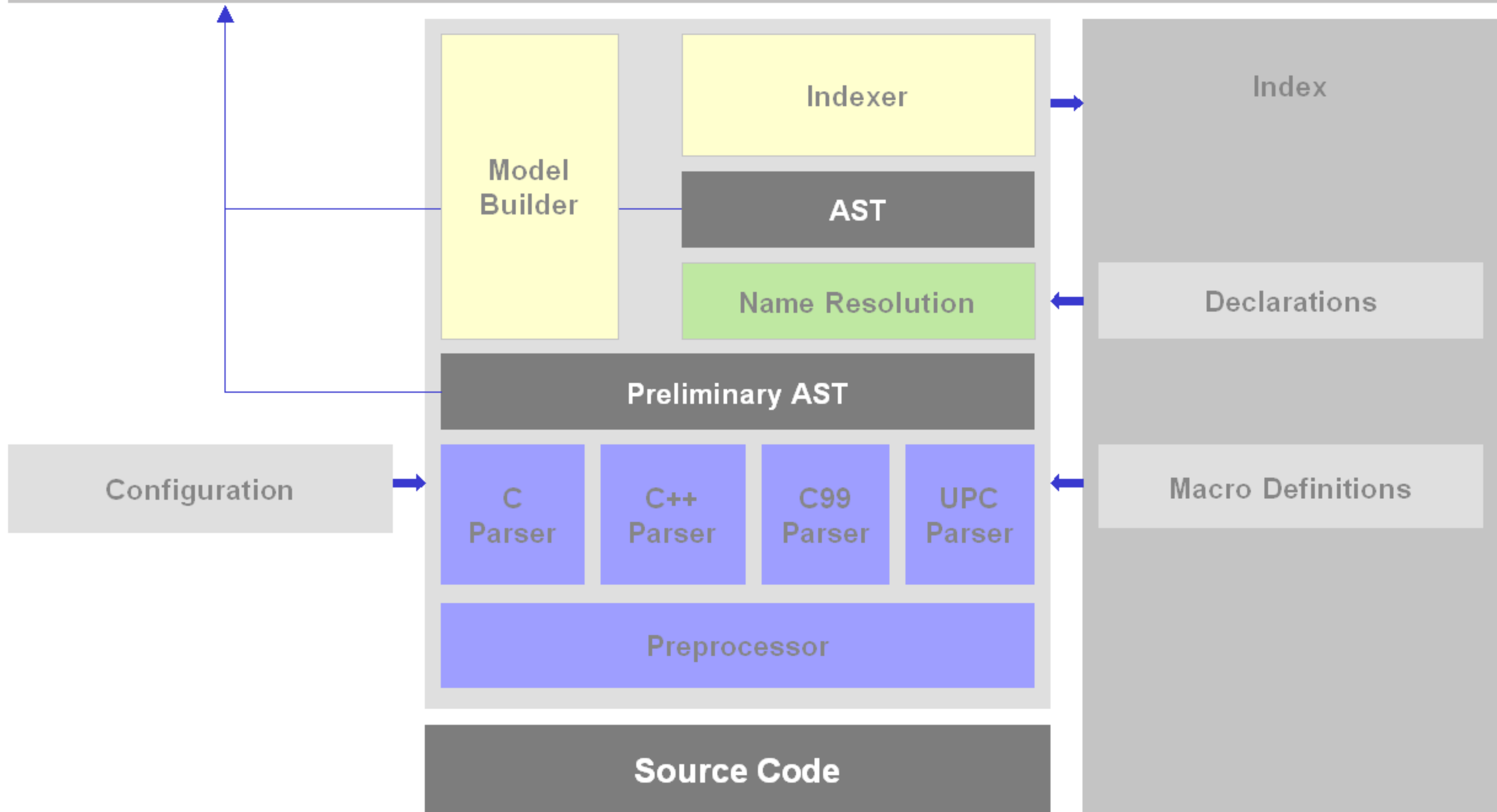


Image courtesy of Marcus Schorn http://wiki.eclipse.org/Image:Parser-arch.png

C/C++ Challenges

- Preprocessor
  - Extra phase between the lexer and the parser
  - Does not have proper imports, instead uses the archaic text based #include directive
  - Macros, Conditional Compilation, Includes
- C++ is very difficult to parse
  - Not LALR(n) for any n
  - Rife with ambiguities and subtleties
- Difficult language constructs
  - Multiple inheritance
  - Templates
  - etc...

© 2012 IBM Corporation; made available under the EPL v1.0

C/C++ Challenges

- Two languages to deal with, C and C++
  - C is **not** a proper subset of C++
- Can't always tell which language to use from the file extension
  - .h file could be C or C++
- Every C and C++ compiler has is own intricacies
  - Slightly different dialects
- Supporting language extensions
  - UPC for example

- Performance!

© 2012 IBM Corporation; made available under the EPL v1.0

eclipse

Editor Framework

- Editor should update its presentation and other related views in real time.
- But, we don't want to re-parse the code in the editor on every keystroke.

- Reconciler thread
  - Maintains a countdown timer (very short, ~3 seconds). Every time you type a character the timer is reset to zero. When the timer expires a “reconcile event” is fired.
  - CDT listens for the reconcile event and re-parses the code in the editor
  - Processes the AST and updates all the views

- Still, parser needs to be fast!

© 2012 IBM Corporation; made available under the EPL v1.0

C Pre-processor (Cpp)

"In retrospect, maybe the worst aspect of Cpp is that it has stifled the development of programming environments for C. The anarchic and character-level operation of Cpp makes nontrivial tools for C and C++ larger, slower, less elegant, and less effective than one would have thought possible."

- Bjarne Stroustrup in The Design and Evolution of C++

© 2012 IBM Corporation; made available under the EPL v1.0

Preprocessor

- An extra phase that runs before the parser

- Include directives

```
#include <stdio.h>
```

  - Replace the #include directive with the contents of the file stdio.h
  - Usually used to include “header” files (that contain only declarations)

- Macros

```
#define max(x,y) (x) > (y) ? (x) : (y)
```

- Conditional compilation

```
#ifdef M
    // some code
#else
    // other code
#endif
```


© 2012 IBM Corporation; made available under the EPL v1.0

eclipse

Preprocessor – Huge problem for accuracy

- Completely text based, no relation to C++ whatsoever...
  - Directives can be inserted literally anywhere

```
static
#ifdef M
int
#else
long
#endif
foo() {
}
```

- In this example conditional compilation directives break up a declaration

- What you see in the editor and what the parser sees are two different things.
  - Disconnect that doesn't happen with other languages like Java

© 2012 IBM Corporation; made available under the EPL v1.0

eclipse

Preprocessor – Huge problem for performance

- A “Translation Unit” is assembled from multiple source files
- A seemingly simple file can become huge after the preprocessor runs.

- helloworld.c

```
#include <stdio.h>
int main() {
  printf("Hello World\n");
}
```

- `$ gcc -E helloworld.c | wc -l`
  - `939`

- 4 lines of code blows up into 939 lines!

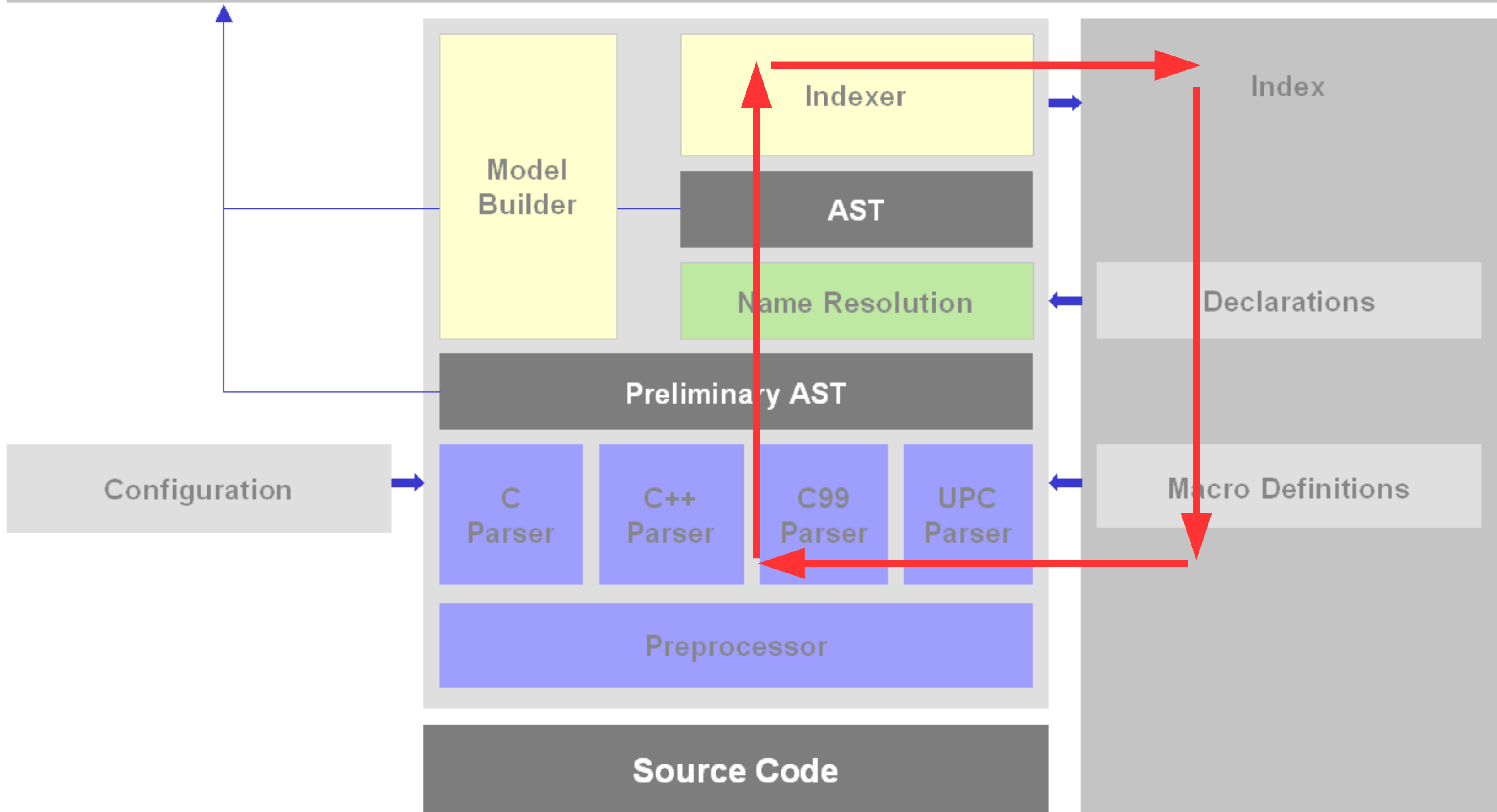
© 2012 IBM Corporation; made available under the EPL v1.0

eclipse

Preprocessor

- For performance the CDT parser will skip parsing of #include directives whenever it can.
- Any macros in the header files that are skipped are still needed for an accurate parse.
  - Get them from the index!

© 2012 IBM Corporation; made available under the EPL v1.0



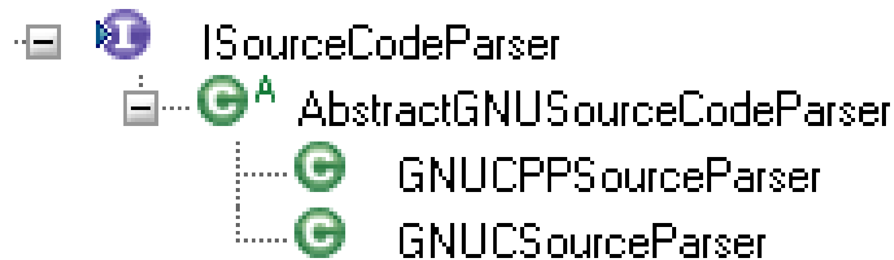
Parser based Tooling
Highlighting, Folding, Formatting, Hovers
Navigation, Content Assist, Search
Include Browser, Call Hierarchy, Type Hierarchy, Outline View
Refactoring
AST
CModel
Index API
Model Builder
Indexer
AST
Name Resolution
Preliminary AST
Index
Declarations
Macro Definitions
Configuration
C Parser
C++ Parser
C99 Parser
UPC Parser
Preprocessor
Source Code

Parsing C/C++

- The parser's job is to convert concrete syntax to abstract syntax
- In other words: convert a char[] into a data structure called an Abstract Syntax Tree (AST)

- CDT supports two languages: C and C++
  - C is not a strict subset of C++, but they do have a lot in common

- Parsers are hand-written recursive descent
- Two parsers, for C and C++, with a common abstract superclass



© 2012 IBM Corporation; made available under the EPL v1.0

eclipse

Ambiguities

```
x * y;
```

- Meaning depends on how x and y have been previously declared
- Could be x multiplied by y

```
int x, y;
x * y;
```

- Could be declaration of a pointer variable y of type x;

```
typedef int x;
x *y;
```


© 2012 IBM Corporation; made available under the EPL v1.0

Ambiguities – The Lexer Feedback Hack

- Well known technique used by compilers, but not by CDT
- Maintain a symbol table during the parse
  - When a declaration is parsed enter the declaration into the symbol table
- Allow the lexer to have access to the symbol table
- When the lexer recognizes an identifier it checks the symbol table to see if the identifier has been previously declared as a type
  - If it has return a *typedef-name* token
  - Otherwise return a normal *identifier* token
- Grammar rules that expect types use *typedef-name* token instead of identifier token

© 2012 IBM Corporation; made available under the EPL v1.0

Ambiguities – The Lexer Feedback Hack

- Relies on the fact that a normal compiler will evaluate #include directives.
  - The parser will see all the declarations in scope and enter them into the symbol table

- But we can't do that for performance reasons
- Need a different approach

© 2012 IBM Corporation; made available under the EPL v1.0

eclipse

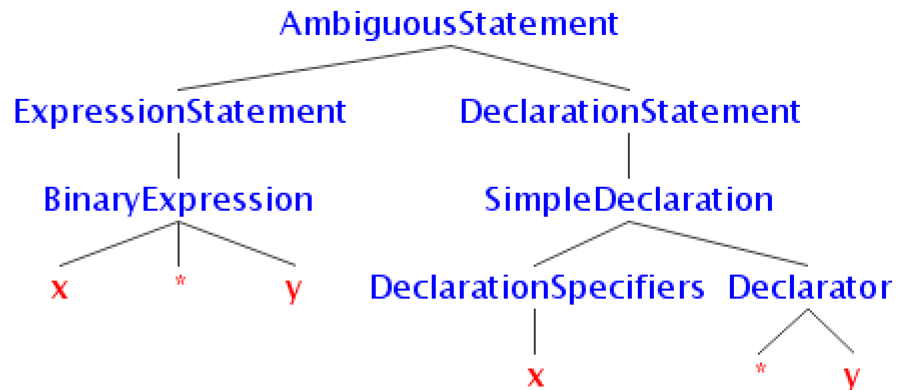
Ambiguities

- Solution used by CDT: backtracking and ambiguity nodes
- Parser knows when it is starting to parse something that may be ambiguous
  - Result of exhaustive analysis of grammar
- Does an initial parse looking for one possibility
- Backtracks and re-parses the same tokens looking for the other possibility
- If both parses succeed create an *ambiguity node* in the AST
  - Ambiguity node contains a list of sub-trees for each possibility
- AST with ambiguity nodes is called the *preliminary AST*

© 2012 IBM Corporation; made available under the EPL v1.0

Ambiguities

```
x * y;
```

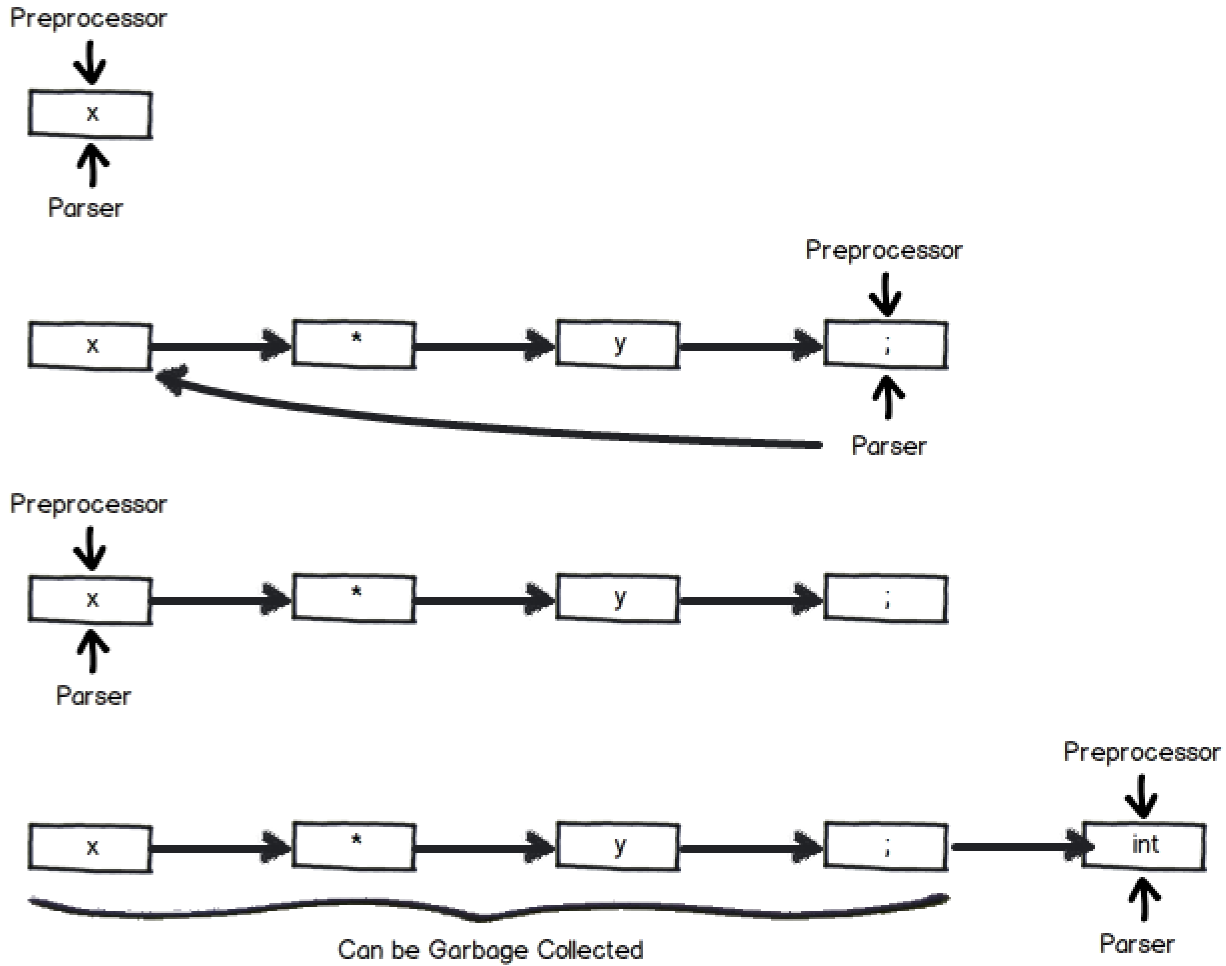


© 2012 IBM Corporation; made available under the EPL v1.0

Backtracking

- The preprocessor runs in tandem with the parser
- Parser calls `preprocessor.fetchToken()` when it wants to see the next token.
  - This causes the preprocessor to recognize the next token
  - But, the preprocessor checks if the token is a #directive or macro name
  - This can cause one token to expand into many tokens
- Preprocessor maintains a linked list of tokens
  - If the preprocessor is at the end of the list it will lex the next token, otherwise it advances to the next token in the list and returns it
- When the parser wants to backtrack it resets the preprocessor back to an earlier token
- Want to do this in a way that doesn't cause too many token objects to be kept in memory.

© 2012 IBM Corporation; made available under the EPL v1.0



© 2012 IBM Corporation; made available under the EPL v1.0

Ambiguity Resolution

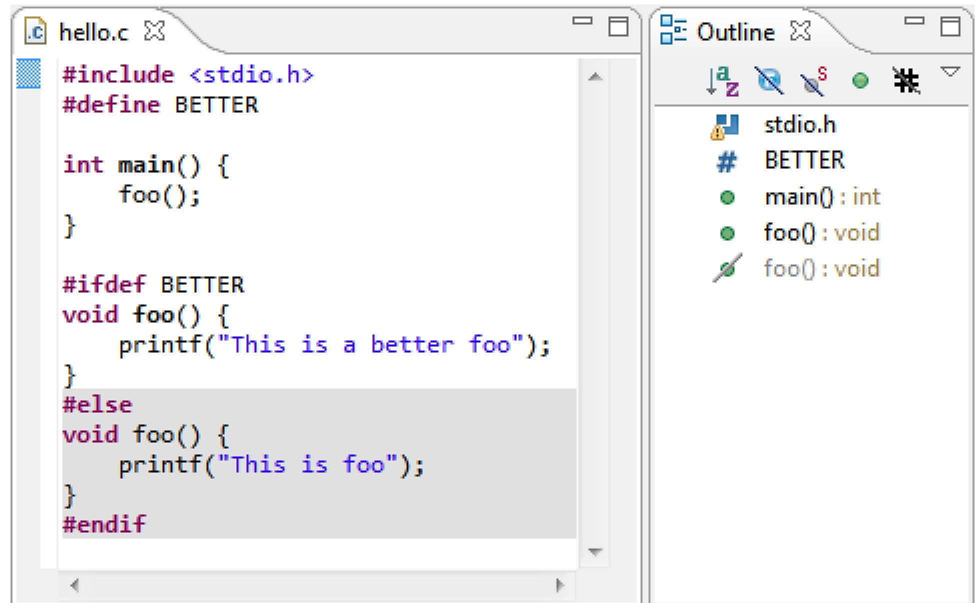
- Job of ambiguity resolution is to pick the correct sub-tree and discard the other ones.

- Algorithm is simple.
  - For each sub-tree
    - Resolve each identifier in context
      - Eg: x * y, look for variables x and y to bind.
  - Keep the sub-tree that has the least number of binding errors
  - If there is a tie, keep the first one
    - There is a rule in C++ that if both possibilities are valid then choose declarations over expressions.
      - just put declarations first

© 2012 IBM Corporation; made available under the EPL v1.0

Parsing Inactive Code

- Parser will attempt to parse inside of inactive code blocks
- Goes into an exploratory parse mode
  - Only parses declarations, skips over function bodies
  - If the parse goes awry parsing of the block is aborted



© 2012 IBM Corporation; made available under the EPL v1.0

AST + Location Map

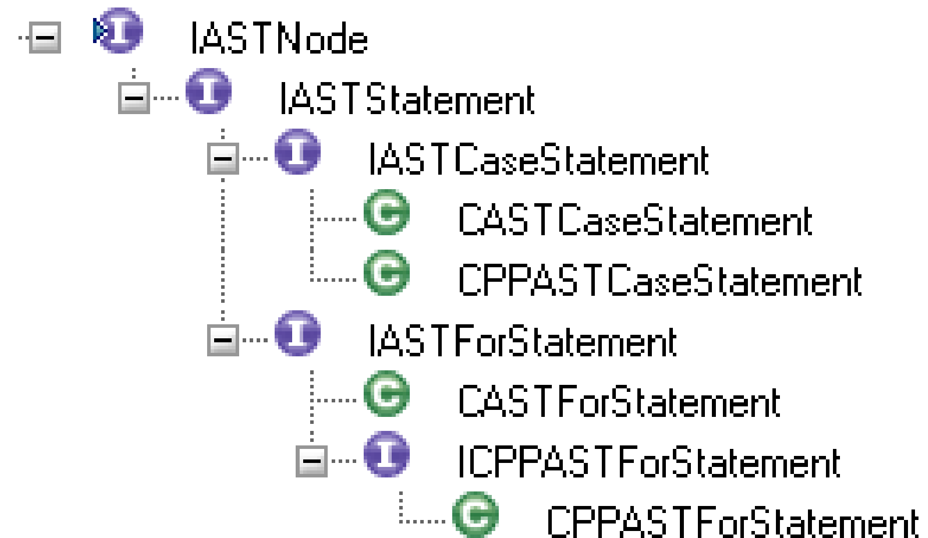
- Attached to the AST is a data structure called the *Location Map*
- Created by the preprocessor
  - Records all the substitutions performed by the preprocessor
- Directly used by the Macro Expansion Hover feature

- Each AST node has offset and length fields
- These are PPP offsets: Post-pre-processor
  - IE offsets into the token stream, not offsets into the original source
  - Location map is a function from (ppp-offset) -> (original-offset, original-file)
- Navigate to a node that comes from a macro
  - Editor will highlight the macro

© 2012 IBM Corporation; made available under the EPL v1.0

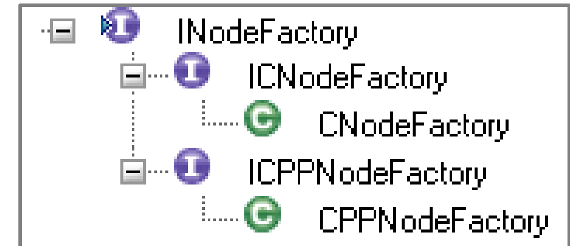
AST

- The AST represents the structure of the source code
- Much of the functionality of CDT editor is based on the AST
- AST node classes for C and C++ are kept in separate packages
  - ~90 node classes for C++
  - ~60 node classes for C
- Implement common interfaces
  - Some algorithms depend on the specific type: semantic analysis
  - Some algorithms only need the interfaces: outline view



© 2012 IBM Corporation; made available under the EPL v1.0

Building The AST – Abstract Factory Pattern




```
abstract class AbstractGNUSourceCodeParser {
    private INodeFactory nodeFactory;
    public AbstractGNUSourceCodeParser(INodeFactory nodeFactory) {
        this.nodeFactory = nodeFactory;
    }
    protected IASTStatement parseWhileStatement() {
        IASTExpression condition = // parse condition
        IASTExpression body = // parse body
        IASTWhileStatement whileStatement = nodeFactory.newWhileStatement(condition, body);
        return whileStatement;
    }
}

interface INodeFactory {
    public IASTWhileStatement newWhileStatement(IASTExpression condition, IASTStatement body);
}
interface ICNodeFactory extends INodeFactory {}
interface ICPPNodeFactory extends INodeFactory {}

class CNodeFactory implements ICNodeFactory {
    public IASTWhileStatement newWhileStatement(IASTExpression condition, IASTStatement body) {
        return new CASTWhileStatement(condition, body);
    }
}

class CPPNodeFactory implements ICPPNodeFactory {
    public IASTWhileStatement newWhileStatement(IASTExpression condition, IASTStatement body) {
        return new CPPASTWhileStatement(condition, body);
    }
}
```


© 2012 IBM Corporation; made available under the EPL v1.0

Building the AST – Abstract Factory

- Advantages
  - Cleaner implementation
    - Code for creating nodes was moved from the parser classes to separate factory classes.
  - Factories are reusable outside of the parser.
    - Used by the CDT refactoring framework.
      - IASTTranslationUnit.getNodeFactory()
    - Used by 3rd party parsers.
      - The UPC parser uses the C node factory.
  - Its easy to add new factory implementations in the future.
    - For example if ObjectiveC support ever gets added to CDT

© 2012 IBM Corporation; made available under the EPL v1.0

Visitor Pattern

- Design pattern used for tree traversal of the AST
- Don't want to add code for each feature directly to the AST classes
  - Don't want code for various features mixed together in the node classes.
  - Want a standard easy-to-use API for processing the AST
  - 3rd parties want to write plug-ins that process the AST.
- Tree traversal the hard way
  - Each AST node has several getX() methods to access child nodes
    - This can be a cumbersome way to traverse the tree
- We want to decouple the data from the operations that process the data.

© 2012 IBM Corporation; made available under the EPL v1.0

Visitor Pattern

- Create a visitor object
  - Must extend `ASTVisitor`
  - `ASTVisitor` has several overloaded `visit(IASTXXX)` methods for each node type
  - Override the visit methods for the node types that you care about
- Each node class has an `accept(ASTVisitor)` method (defined in `IASTNode`)
  - Calls `visit(this)`

IASTNode
- accept(ASTVisitor) : boolean
- getChildren() : IASTNode[]
- getParent() : IASTNode
- getTranslationUnit() : IASTTranslationUnit

ASTVisitor
- PROCESS_ABORT : int
- PROCESS_CONTINUE : int
- PROCESS_SKIP : int
- shouldVisitDeclarations : boolean
- shouldVisitExpressions : boolean
- shouldVisitNames : boolean
- shouldVisitStatements : boolean
- visit(IASTDeclaration) : int
- visit(IASTExpression) : int
- visit(IASTName) : int
- visit(IASTStatement) : int

© 2012 IBM Corporation; made available under the EPL v1.0

AST Visitor

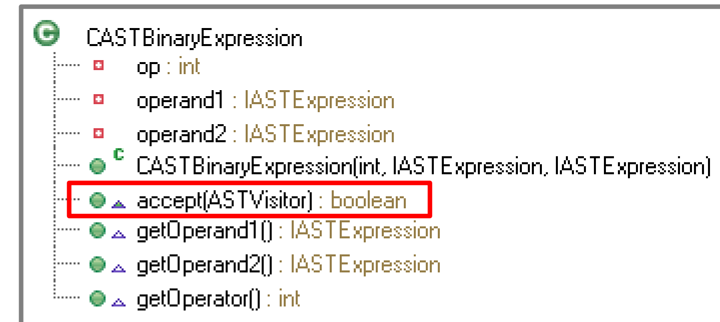
- Example of an accept method in a node that has children.

```
public boolean accept( ASTVisitor action ){

    if( action.shouldVisitExpressions ){
        switch( action.visit( this ) ){
            case ASTVisitor.PROCESS_ABORT : return false;
            case ASTVisitor.PROCESS_SKIP  : return true;
            default : break;
        }
    }

    if (operand1 != null && !operand1.accept(action))
        return false;
    if (operand2 != null && !operand2.accept(action))
        return false;


    return true;
}
```



© 2012 IBM Corporation; made available under the EPL v1.0

AST Visitor

- Example of a simple visitor that collects all the name nodes (identifiers) in the AST

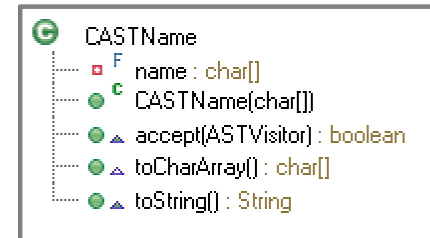
```
public List<IASTName> collectAllNames(IASTTranslationUnit tu) {
    final List<IASTName> names = new ArrayList<IASTName>();

    ASTVisitor visitor = new ASTVisitor() {
        { shouldVisitNames = true; }

        @Override
        public int visit(IASTName name) {
            names.add(name);
            return PROCESS_CONTINUE;
        }
    };

    tu.accept(visitor);

    return names;
}
```




```
public boolean accept(ASTVisitor action) {
    if (action.shouldVisitNames) {
        switch (action.visit(this)) {
        case ASTVisitor.PROCESS_ABORT:
            return false;
        case ASTVisitor.PROCESS_SKIP:
            return true;
        default:
            break;
        }
    }
    return true;
}
```


© 2012 IBM Corporation; made available under the EPL v1.0

Tree Traversal

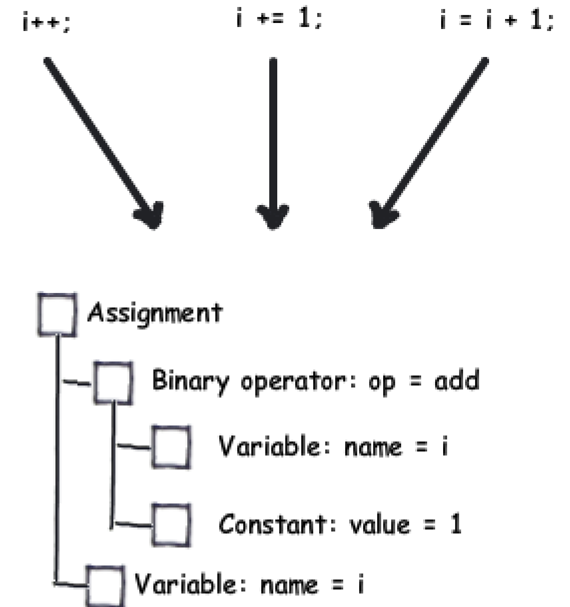
- Tree traversal, two variations
  - 1) The visitor controls the traversal order → visitor calls accept()
    - More flexible, supports complex traversals
    - Requires more code in the visitor
    - Traversal code may end up duplicated in each visitor
  - 2) The AST controls the traversal order → nodes call accept()
    - Less flexible
    - Easier to implement the visitor if the standard traversal order is acceptable.

- CDT uses option 2
  - A depth-first traversal order is hard-coded into the AST
    - This is by far the most common traversal order
  - The API does provide some control over what nodes to visit
  - It is still possible to write a visitor that has complete control over traversal order.

© 2012 IBM Corporation; made available under the EPL v1.0

Desugaring

- Syntactic Sugar
  - Syntax that is equivalent to some other syntax in the language but is more convenient or compact.
    - i++;
    - i += 1;
    - i = i + 1;
- Desugaring
  - The parser produces the same AST fragment
  - Convenient for code generation.
- AST produced by IDE cannot be desugared.
  - The AST needs to represent exactly what is in the user's source.

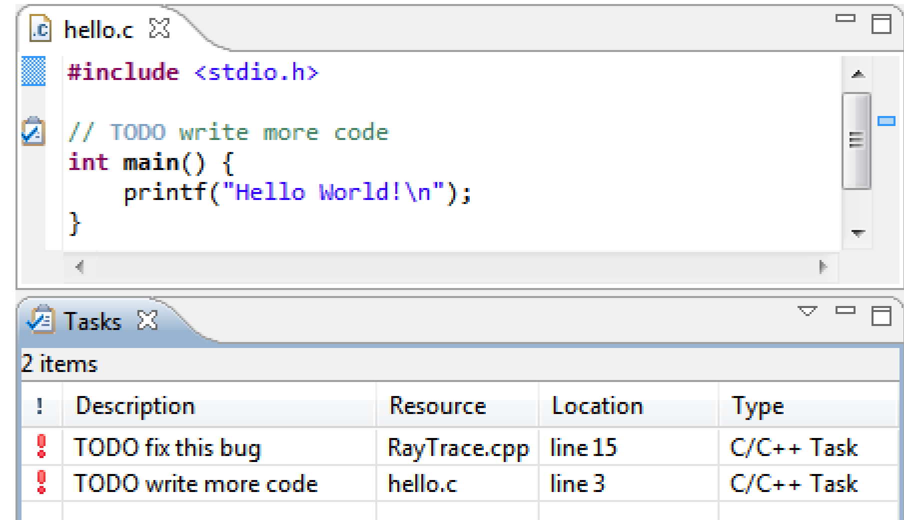


© 2012 IBM Corporation; made available under the EPL v1.0

Comments

- Comments are preserved in the AST
- Available as a flat list of “comment nodes”

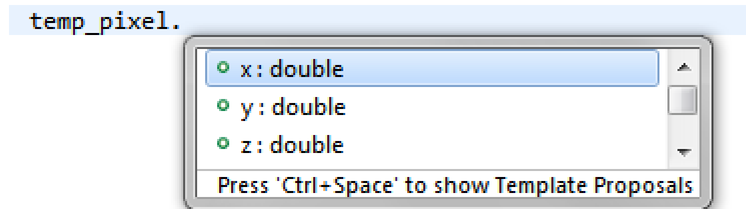
- Refactorings that move code around need to move the comments too
- Special comments are recognized
  - TODO comments
    - Stored in the index



© 2012 IBM Corporation; made available under the EPL v1.0

Content Assist

- User can type part of a statement and then get a list of possible completions.



- The user has not finished typing the statement
  - This is a syntax error!
- Parser must:
  - Recover from the syntax error
  - Unwind the parse stack
  - Return a “completion node”, used to compute the list of proposals

© 2012 IBM Corporation; made available under the EPL v1.0

Content Assist – The Preprocessor's Job

- Two special types of tokens
  - *Completion* token
  - *End-of-completion* token

- The offset of the cursor position is given to the preprocessor
- When the offset is reached the preprocessor returns a *Completion* token
  - Similar to an identifier token
  - If the user typed part of an identifier the token will contain this text
- Stops processing the input character stream and starts indefinitely returning *End-of-completion* tokens

© 2012 IBM Corporation; made available under the EPL v1.0

eclipse

Content Assist – The Parser's Job

- Parser will accept a *Completion* token anywhere an identifier token would be legal
- The only difference is that a Completion token alerts the parser to generate an extra side-effect: creating a completion node
- Will backtrack and re-parse to cover potential ambiguities.
  - Will get a completion node for every possibility

- *End-of-completion* tokens allow the parser to complete successfully

- An End-of-completion token will match punctuation that can be used to end statements and close expressions and scopes, including semi-colons, closing parenthesis, closing braces and others

© 2012 IBM Corporation; made available under the EPL v1.0

Content Assist - Example

```
int s = sizeof(f<ctrl-space>
```

- preprocessor produces the following token stream

```
int, identifier, assign, sizeof, left-paren, completion,
end-of-completion, end-of-completion...
```

- which the parser will interpret as

```
int, identifier, assign, sizeof, left-paren, identifier,
right-paren, semi-colon
```


© 2012 IBM Corporation; made available under the EPL v1.0

The End

- References
  - CDT Project home page: http://eclipse.org/cdt/
  - CDT Wiki: http://wiki.eclipse.org/CDT
  - Download eclipse and CDT: http://www.eclipse.org/downloads/
  - Lexer Feedback Hack: http://en.wikipedia.org/wiki/The_lexer_hack

© 2012 IBM Corporation; made available under the EPL v1.0