Extensible Parsers - C99/UPC Parsers

Mike Kucera
Jason Montojo
IBM Eclipse CDT Team

Overview

- Goals
  - To create a parser framework that allows language extensions to be easily added to CDT
  - Modularity
  - Clean implementation, maintainability
  - Performance

- Support for Unified Parallel C (UPC) needed by the Parallel Tools Project
  - UPC spec is an extension to the C99 spec

C99 Parser in CDT 4.0

- C99 parser base
  - Designed to be extensible
- UPC parser
  - Built on top of C99 parser

Language Extensibility in CDT

- What CDT currently provides
  - Extension point for adding new parsers
  - Map languages to content types
  - Syntax highlighting can be extended to new keywords
  - Add new types of AST nodes
- What CDT does not provide
  - A parser that can be directly extended to support new syntax
  - A reusable preprocessor

C99 Preprocessor

- New preprocessor written from scratch
- Much cleaner implementation than DOM preprocessor
  - DOM preprocessor:
    - Lexing and preprocessing are combined, not modular
    - Processes raw character stream, very complex code to do this
    - Doesn't handle comments properly
  - C99 Preprocessor:
    - Lexing and preprocessing are separated, modular
    - Token based, input is first lexed into tokens then fed to preprocessor, much cleaner
    - Comments are correctly resolved by the lexer

C99 Parser in CDT 4.0

- Different approach than the DOM parser
  - DOM parser completely hand written
- C99 Parser generated from grammar files using a parser generator
  - Using LPG - LALR Parser Generator
  - Bottom-up parsing approach
  - Grammar file looks similar to the spec
- Some parts of DOM parser are reused
  - AST
  - LocationMap

LPG – LALR Parser Generator

- Two parts
  - The generator (lpg.exe)
    - Generates parse tables from grammar file
    - Parse tables are basically a specification of a finite state machine
  - The runtime (java library)
    - Contains the parser driver and supporting classes
    - Parser driver interprets the parse tables

LPG – LALR Parser Generator

- LPG is used by several eclipse projects including:
  - Model Development Tools (MDT)
  - Graphical Modeling Framework (GMF)
  - Generative Modeling Technologies (GMT)
  - Data Tools Platform (DTP)
  - SAFARI
  - Java Development Tools (JDT, in the bytecode compiler)
- Part of Orbit project

LPG – Benefits

- Automatic
  - Computation of AST node offsets
  - Backtracking
  - Syntax error recovery
- Clean separation of parser and the code that builds the AST
- Grammar file inheritance
  - Source of parser extensibility

C99 Grammar File Example

```
statement
    ::= labeled_statement
      | compound_statement
      | expression_statement
      | selection_statement
      | iteration_statement
      | jump_statement
      | ERROR_TOKEN
          /.$ba  consumeStatementProblem();  $ea./


iteration_statement
    ::= 'do' statement 'while' '(' expression ')' ';'
          /.$ba  consumeStatementDoLoop();  $ea./

      | 'while' '(' expression ')' statement
          /.$ba  consumeStatementWhileLoop();  $ea./

      | 'for' '(' expression ';' expression ';' expression ')' statement
          /.$ba  consumeStatementForLoop(true, true, true);  $ea./
```

AST Building Actions

```
/**
 * iteration_statement ::= 'while' '(' expression ')' statement
 */
public void consumeStatementWhileLoop() {

    IASTWhileStatement whileStatement = nodeFactory.newWhileStatement();

    IASTStatement  body      = (IASTStatement)  astStack.pop();
    IASTExpression condition = (IASTExpression) astStack.pop();


    whileStatement.setBody(body);
    body.setParent(whileStatement);
    body.setPropertyInParent(IASTWhileStatement.BODY);

    whileStatement.setCondition(condition);
    condition.setParent(whileStatement);
    condition.setPropertyInParent(IASTWhileStatement.CONDITIONEXPRESSION);

    setOffsetAndLength(whileStatement);

    astStack.push(whileStatement);
}
```

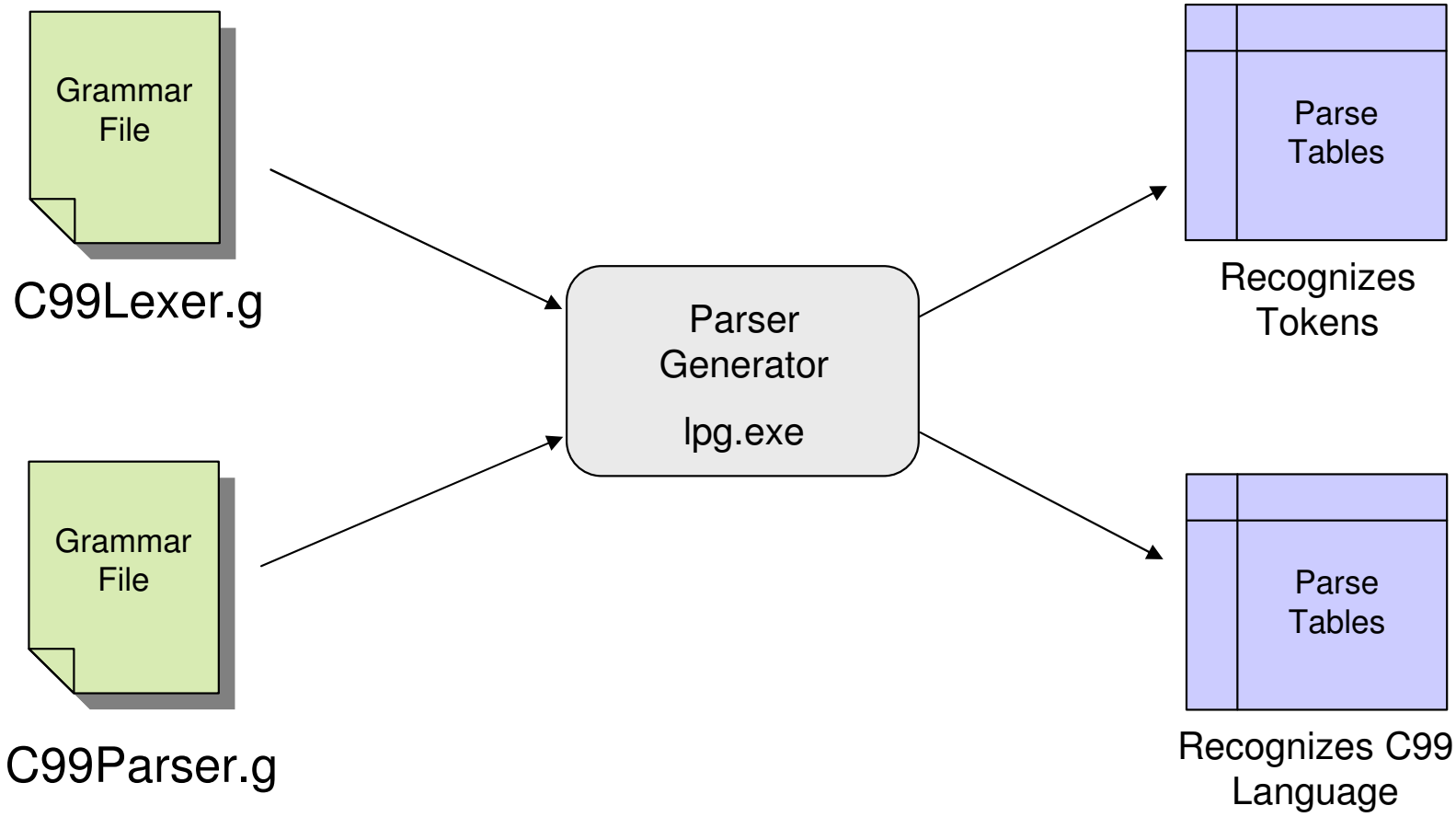
Content Assist

- 5 simple grammar rules

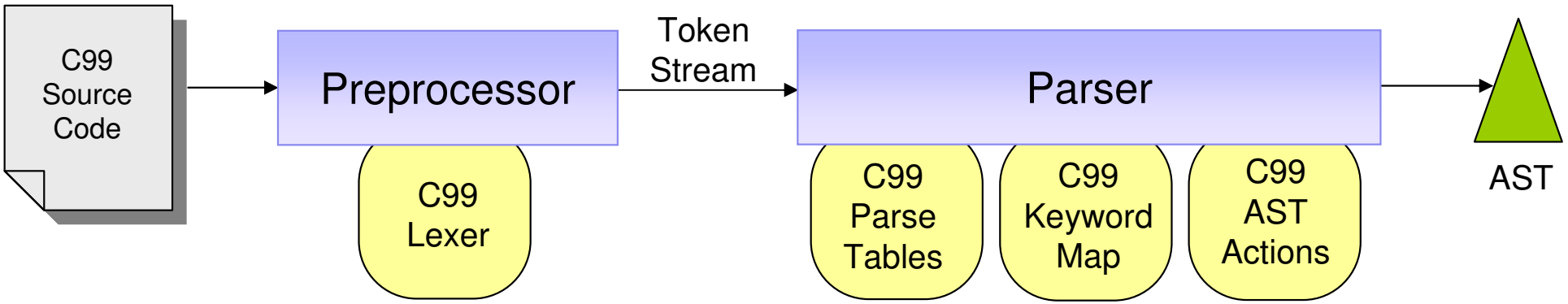
```
ident ::= 'identifier' | 'Completion'

']' ::=? 'RightBracket' | 'EndOfCompletion'
')' ::=? 'RightParen'   | 'EndOfCompletion'
'}' ::=? 'RightBrace'   | 'EndOfCompletion'
';' ::=? 'SemiColon'    | 'EndOfCompletion'
```

- First rule says that a Completion token can occur anywhere an identifier token can occur.
- Next 4 rules allow the parse to complete successfully after a Completion token has been encountered.

Generating The Parser From Grammar Files



Architecture of C99 Parser



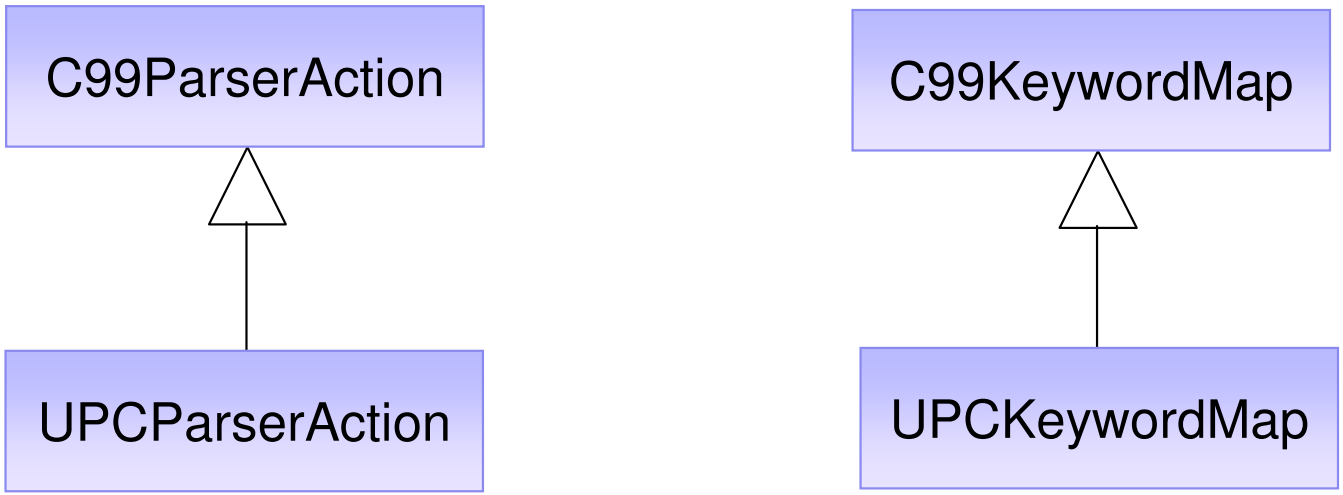
Extensibility – Supporting UPC

- UPC grammar file extends the C99 grammar file
  - Adds new grammar rules for UPC syntax
  - Generates new parse tables that recognize UPC

```
$Import
   C99Parser.g
$End


iteration_statement
    ::=  'upc_forall' '(' expression ';' expression ';' expression ';'
                          affinity ')' statement
          /.$ba  consumeStatementUPCForallLoop(true, true, true, true); $ea./
```

Extensibility – Supporting UPC

- Extend C99 classes.

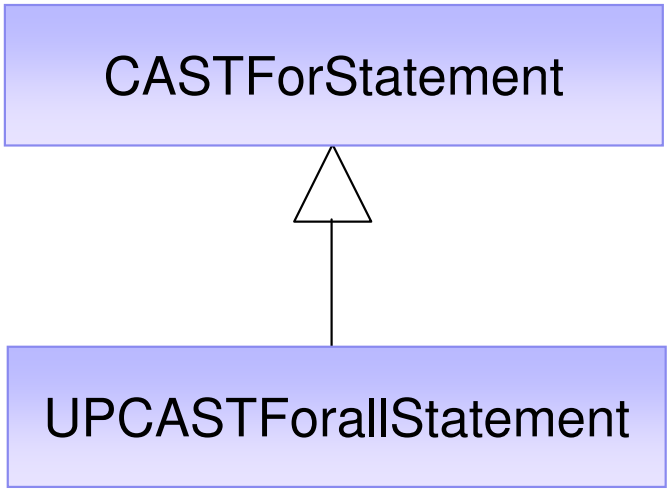


Adds actions for new grammar rules

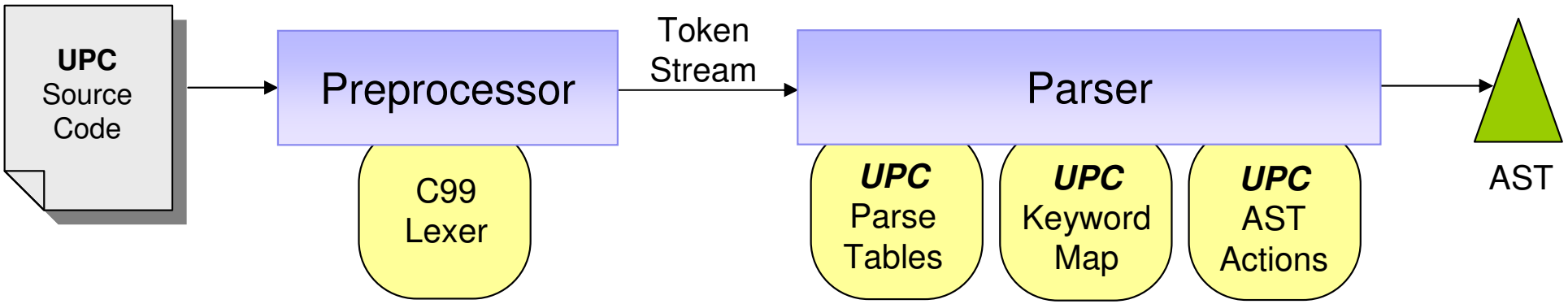
Adds mappings for new UPC keywords like ‘upc_forall’

Extensibility – Supporting UPC

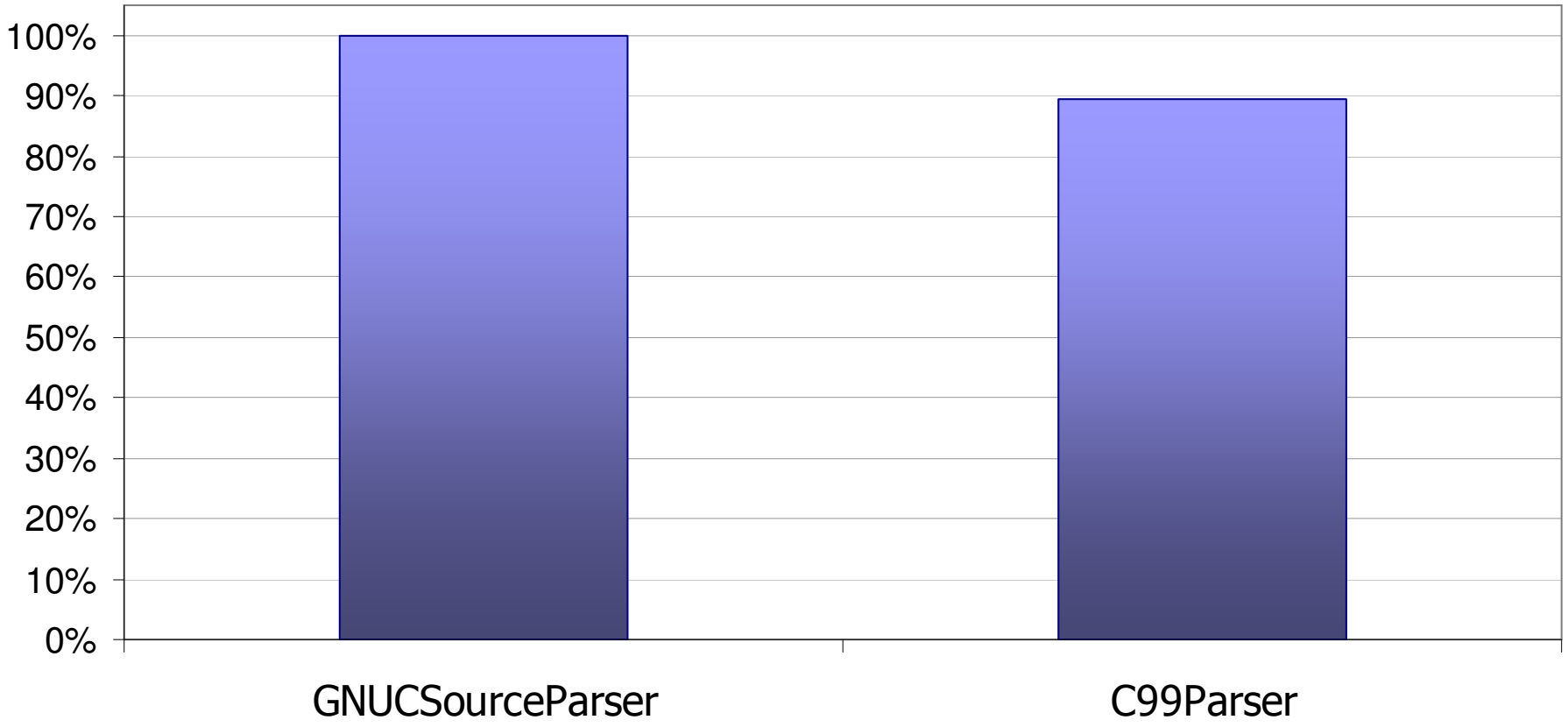
- Create AST node classes for new language constructs



Architecture of UPC Parser



Performance



Future Work

- Make the preprocessor reusable
  - Reusable on any token stream
  - Use for FORTRAN etc…
- Support for C++
  - Advanced approach
- Provide compiler specific extensions
  - GCC, XLC etc…
- Further performance enhancements
  - We haven't spent much time on optimizations yet