# Extending PCCTS:
# An Object Architecture

- Introduction
- Motivation
- Technical Overview
- Sample Applications
- Open Issues

# Introduction

- Runtime Compiler Construction Environment (RCCE)
- Proposal, Not a Design Spec
  - very little code has been written
  - still lots of open questions
- Plan
  - use C++ (lots of demand)
  - 1-3 person years + Guru (Terence?)
- Looking for Feedback
  - would RCCE help you?
  - critical feedback
  - development assistance?

# Motivation

- provide object model for PCCTS
  - unify interfaces
    - » grammar, actions, etc.
    - » lookahead, parsing alogrithms
    - » symbol table management
    - » translation
  - extensibility and simplicity (?)
- provide runtime interfaces
  - change or extend a grammar at runtime
  - use C++ classes, inheritance, and exceptions
  - simplifies build process
  - smaller executables (?)
- more ...

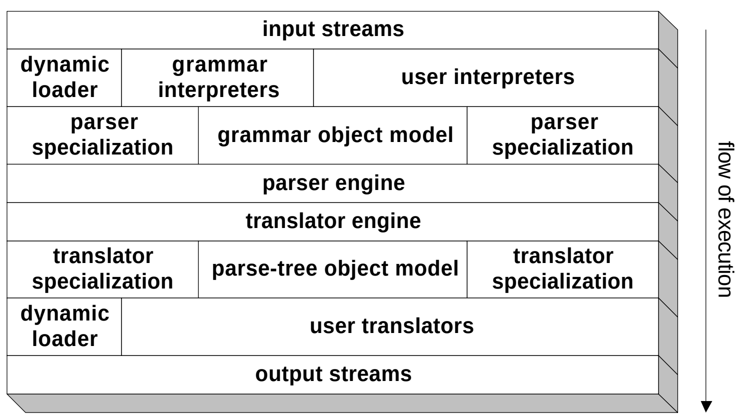
# Motivation

- built-in parsing facilities
  - symbol table management
  - translation
- simplifies construction of “killer app”
  - GUI-based compiler construction apps
  - grammars (including RCCE) are resident
  - can be changed, extended at runtime

# Technical Overview

- RCCE Architecture
- Object Model for Grammar Nodes
- Grammar Node Overview
- Lookahead and Match Strategies
- Passing Values With Attribute Tables
- Actions and Dynamic Loaders
- Symbol Table Management
- Parse Trees/Translation
- Input and Output Streams

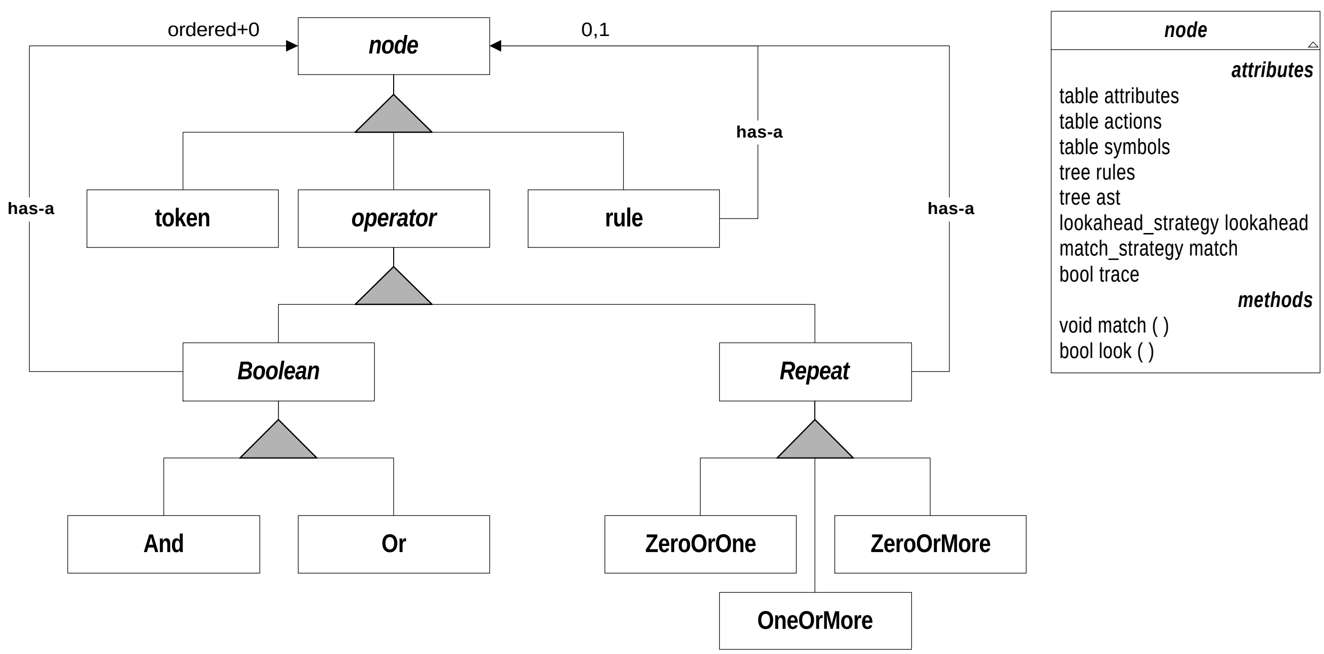
# RCCE Architecture



# RCCE Architecture

- Data comes in on an **input stream**. Each stream is dynamically wired to one of potentially several resident **interpreters**. The **grammar** may use a **specialization** of the generic **parser engine**. Specializations might include **lookahead** and **matching** strategies and can be specified on a per node basis.
- **Actions** can be executed by association with both grammar and **parse-tree** nodes. By default, a parse-tree and **symbol table** are maintained. These can also be specialized. New actions can be˝ loaded using a **dynamic loading** mechanism, such as DLL entry points or runtime loading of an object module.
- At any point, a parse-tree can be executed. Optionally, it˝ may be dynamically wired to an **output stream**. It might also simply manipulate the runtime environment. The former would be more useful for a compiler, the latter for modifying the grammar˝ or parse-tree. These are not mutually exclusive. For example, the˝ output for a grammar parse-tree might be a binary image of the grammar.

# Object Model for Grammar Nodes



# Lookahead and Match Strategies

- specialized at compile-time
  - bool lookahead ( input_stream )
- match returns parse-tree
  - parse_tree match ( input_stream )
- use template/hook methods
  - template methods parameterize fundamental PCCTS parsing algorithms
  - hook methods allow subclasses to modify the algorithms

# Lookahead and Match Strategies

- **match throws a C++ exception with a copy of specific node where match failed**
- **each node can have a different match strategy**

| ***lookahead_strategy*** |
|---|
| *bool lookahead ( input_stream )* |
| *bool before_node( node )* |
| *bool after_node( node )* |
| *int depth()* |
| *int approx()* |

| ***match_strategy*** |
|---|
| *parse_tree match ( input_stream )* |
| *throw no_match* |
| *void before_node( node )* |
| *void after_node( node )* |

# Passing Values With Attribute Tables

- each node maintains a set of key-value pairs
- whenever an action or other event occurs, these values can be modified
- emulates PCCTS multivalued call-return semantics
- open question: mechanism for passing attribute tables up or down the parse-tree

# Actions and Dynamic Loaders

- action table would store various kinds of ‘handlers’
  - init, before_match, before_look, after_match, after_look, no_match
- actions would be loaded at either runtime (for built-in actions) or dynamically using native loader facilities
- an action-scripting language could be provided that translates to object code and then dynamically loads the object code

# Parse Trees/Translation

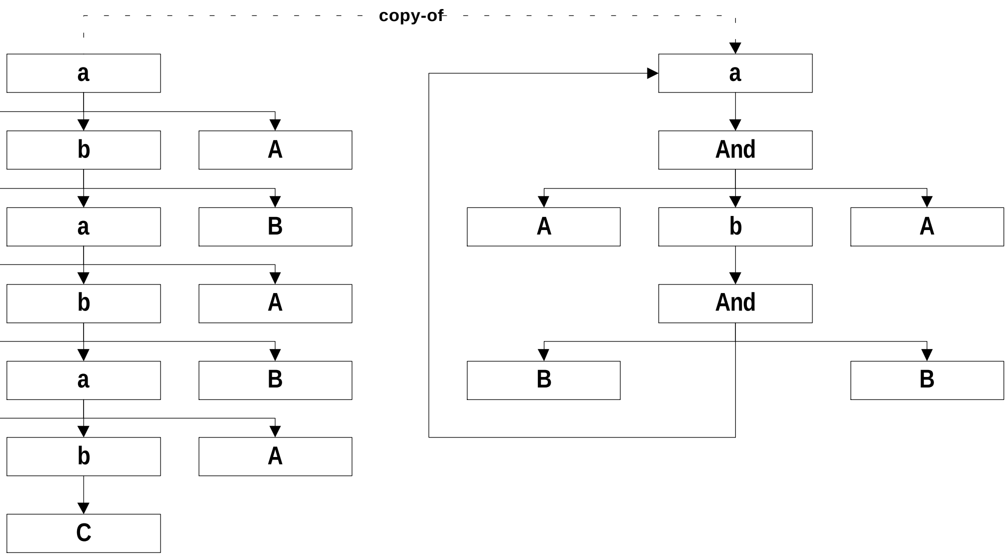
- uses token subclass of node along with rule
- printer objects connect parse trees to output streams
- “execute” parse trees, which calls printer object for each node
- printer object uses translation tree for each rule to determine format of output
- printer objects can be specialized
- open question: how are translations trees employed?

# Parse Trees/Translation

```
a: A b A;
b: B a B | C;

Alternating A's and B's with a C in the
middle.
```

ABABACABABA




**Parse Tree**

**Grammar Tree**

# Symbol Table Management

- simple key-value, where key is a string and value is a symbol object
- each node has a local symbol table
- searches for symbols go up the tree
- new symbol types can be created a runtime
  - just like grammar or parse-tree nodes
  - contain attribute tables
- specializations to the symbol table can be made at compile-time
  - for example, to modify the symbol resolution algorithm

# Input and Output Streams

- builds on standard C++ iostreams
  - portable
  - easy to use
  - wide-character support (ANSI draft)
  - standard streaming model

# Possible Applications

- GUI-based compiler construction
- C++ interpreter/browser

# Open Issues

- performance
  - RCCE and PCCTS could use the same grammar
  - RCCE grammars could then be “dumped” into PCCTS
    - » when high-performance is required
    - » and runtime modifications are not
  - BUT will RCCE’s performance be tolerable?
- passing values between nodes
- format of translator trees
- portability of dynamic loading schemes