Welcome!
Everything is fine.

The Duseltronik

Down the Recursive Descent Rabbit Hole

Overview

- Introduction: Parsing Expression Grammars.
- The PEGTL: Implementing a recursive descent parser in C++.
  - Not a tutorial on how to *use* the PEGTL.
- The Simpletronik: Attaching actions to grammar rules.
- The Duseltronik: All the way down the rabbit hole.
- Using the Duseltronik: Tracer, Parse Tree, …

Parsing Expression Grammars

- Introduced by Bryan Ford in 2004.
- Alternative to Context-Free Grammars.
- Fully deterministic, ordered choice, zero-/one-or-more are truly greedy.
- Direct model of a recursive descent parser.
- Add the *and*- and the *not*-predicate.

Parsing Expression Grammars

Sequence: $e_1\ e_2$

Ordered Choice: e_1 / e_2

Zero-or-More: e*

One-or-More: e+

Option: e?

And-Predicate: &e

Not-Predicate: !e

Terminal symbols: 'a'

Nonterminal symbols: A ← e

Empty String: ε

The PEGTL

Parsing Expression Grammar Template Library

The PEGTL

- Started in 2007 by Dr. Colin Hirsch.
- Parser Combinator approach.
- In-language, no separate translation of grammar to code.
- Unified, consistent definition of rules using classes and templates.
- Atomics, combinators, convenience rules, and custom rules.

```
struct rule
{
   // MAY consume input when it matches, i.e. when it returns true
   // MUST NOT consume input when it does not match, i.e. when it returns false
   template< typename Input >
   static bool match( Input& in );
};
```

```
struct success
{
   template< typename Input >
   static bool match( Input& in )
   {
      return true;
   }
};
```

```
struct eof
{
   template< typename Input >
   static bool match( Input& in )
   {
      return in.empty();
   }
};
```

```
template< char C >
struct one
{
   template< typename Input >
   static bool match( Input& in )
   {
      if( !eof::match( in ) && ( in.peek_char() == C ) ) {
         in.bump();
         return true;
      }
      return false;
   }
};

// one< 'a' >
```

“…, a **parser combinator** is a higher-order function that accepts several parsers as input and returns a new parser as its output.”

–*Wikipedia*

```cpp
template< typename R1, typename R2 >
struct sor
{
   template< typename Input >
   static bool match( Input& in )
   {
      return R1::match( in ) || R2::match( in );
   }
};

// sor< one< 'a' >, one< 'b' > >
// sor< sor< one< 'a' >, one< 'b' > >, one< 'c' > >
```

```
template< typename... Rules >
struct sor
{
   template< typename Input >
   static bool match( Input& in )
   {
      return ( Rules::match( in ) || ... );
   }
};

// sor< one< 'a' >, one< 'b' >, one< 'c' > >
```

```
template< typename... Rules >
struct seq
{
   template< typename Input >
   static bool match( Input& in )
   {
      return ( Rules::match( in ) && ... );
   }
};
```

```
template< typename... Rules >
struct seq
{
   template< typename Input >
   static bool match( Input& in )
   {
      const auto old = in;
      if( ( Rules::match( in ) && ... ) ) {
         return true;
      }
      // MUST NOT consume input when returning false
      in = old;
      return false;
   }
};
```

```
template< typename Rule >
struct opt
{
   template< typename Input >
   static bool match( Input& in )
   {
      Rule::match( in );
      return true;
   }
};
```

```
template< typename Rule >
struct star
{
   template< typename Input >
   static bool match( Input& in )
   {
      while( Rule::match( in ) ) {}
      return true;
   }
};
```

```cpp
template< typename Rule >
struct plus
   : seq< Rule, star< Rule > >
{};
```

```
template< typename Rule >
struct at  // the PEG and-predicate
{
   template< typename Input >
   static bool match( Input& in )
   {
      const auto old = in;
      const bool result = Rule::match( in );
      in = old;
      return result;
   }
};
```

```
template< typename Rule >
struct not_at  // the PEG not-predicate
{
   template< typename Input >
   static bool match( Input& in )
   {
      return !at< Rule >::match( in );
   }
};
```

Convenience Rules

- Atomic rules: `success`, `eof`, `one`, …
- Combinators: `sor`, `seq`, `opt`, `star`, `plus`, `at`, `not_at`.
- Tedious to build grammars with only these atoms and combinators.
- Many pre-defined convenience rules available.
- Often more efficient than their naïve equivalent.

```
template< char... Cs >
struct string
   : seq< one< Cs >... >
{};
```

```
template< char... Cs >
struct string
{
   template< typename Input >
   static bool match( Input& in )
   {
      // more efficient implementation
   }
};
```

```
// example grammar to parse C-style block comments

struct c_begin : string< '/', '*' > {};
struct c_end : string< '*', '/' > {};

struct c_char : sor< one< '\t' >, eol, print > {};

struct comment
   : seq< c_begin, star< seq< not_at< c_end >, c_char > >, c_end >
{};
```

```
// example grammar to parse C-style block comments

struct c_begin : string< '/', '*' > {};
struct c_end : string< '*', '/' > {};

struct c_char : sor< one< '\t' >, eol, print > {};

struct comment
   : seq< c_begin, star< seq< not_at< c_end >, c_char > >, c_end >
{};
```

```
// example grammar to parse C-style block comments

struct c_begin : string< '/', '*' > {};
struct c_end : string< '*', '/' > {};

struct c_char : sor< one< '\t' >, eol, print > {};

struct comment
   : seq< c_begin, until< c_end, c_char > >
{};
```

```
template< typename Condition, typename Rule >
struct until
   : seq< star< seq< not_at< Condition >, Rule > >, Condition >
{};
```

```
template< typename Condition, typename Rule >
struct until
{
   template< typename Input >
   static bool match( Input& in )
   {
      // avoids parsing Condition twice at the end
   }
};
```

```cpp
#include <iostream>
#include <iomanip>

#include <tao/pegtl.hpp>
using namespace tao::pegtl;

struct c_begin : string< '/', '*' > {};
struct c_end : string< '*', '/' > {};
struct c_char : sor< one< '\t' >, eol, print > {};
struct comment : seq< c_begin, until< c_end, c_char > > {};
struct grammar : seq< comment, eof > {};

int main( int argc, char* argv[] )
{
   for( int i = 1; i < argc; ++i ) {
      argv_input in( argv, i );
      std::cout << std::boolalpha << parse< grammar >( in ) << std::endl;
   }
}
```

The Simpletronik

- Getting a boolean result is insufficient.
- Add *state* to store data.
- Add *actions* to a rule to modify the state.

```
template< typename... Rules >
struct sor
{
   template< typename Input >
   static bool match( Input& in )
   {
      return ( Rules::match( in ) || ... );
   }
};
```

```
template< typename... Rules >
struct sor
{
   template< template< typename... > class Action,
             typename Input,
             typename... States >
   static bool match( Input& in, States&&... st )
   {
      return ( Rules::template match< Action >( in, st... ) || ... );
   }
};
```

```
template< typename... Rules >
struct sor
{
   template< template< typename... > class Action,
             typename Input,
             typename... States >
   static bool match( Input& in, States&&... st )
   {
      return ( ( Rules::match< Action >( in, st... ) &&
                 ( Action< Rules >::apply( in, st... ), true ) ) || ... );
   }
};
```

> “All problems in computer science
> can be solved by another level of indirection.”
>
> *–David Wheeler*

```
template< typename... Rules >
struct sor
{
   template< template< typename... > class Action,
             typename Input,
             typename... States >
   static bool match( Input& in, States&&... st )
   {
      return ( pegtl::match< Rules, Action >( in, st... ) || ... );
   }
};
```

```
template< typename Rule,
          template< typename... > class Action,
          typename Input,
          typename... States >
bool match( Input& in, States&&... st )
{
   if( Rule::match< Action >( in, st... ) ) {
      Action< Rule >::apply( in, st... );
      return true;
   }
   return false;
}
```

```
template< typename Rule,
          template< typename... > class Action,
          typename Input,
          typename... States >
bool match( Input& in, States&&... st )
{
   const auto begin = in.current();
   if( Rule::match< Action >( in, st... ) ) {
      const auto end = in.current();
      const action_input< Input > ai( begin, end );
      Action< Rule >::apply( ai, st... );
      return true;
   }
   return false;
}
```

```
template< typename Rule,
          template< typename... > class Action,
          typename Input,
          typename... States >
auto match( Input& in, States&&... st )
  -> std::enable_if_t< has_apply_v< Rule, Action, Input, States... >, bool >
{
   const auto begin = in.current();
   if( Rule::match< Action >( in, st... ) ) {
      const auto end = in.current();
      const action_input< Input > ai( begin, end );
      Action< Rule >::apply( ai, st... );
      return true;
   }
   return false;
}
```

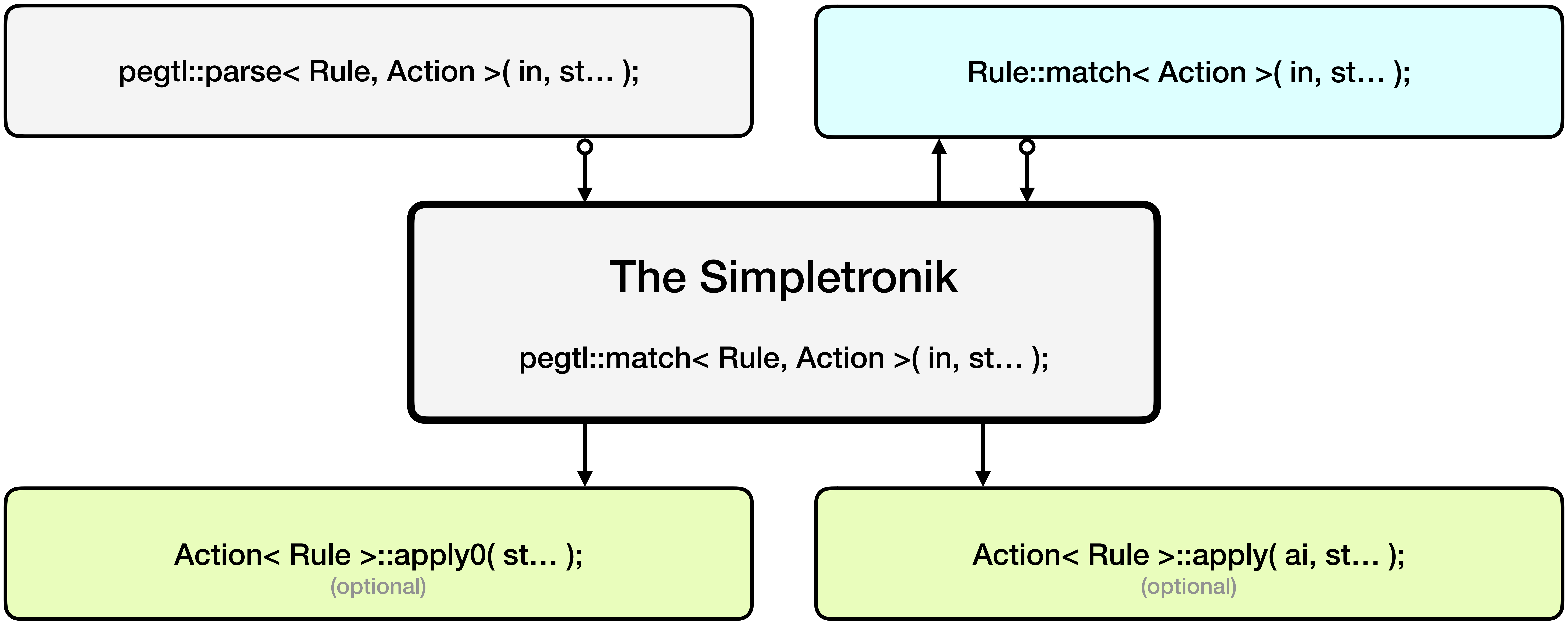
```
template< typename Rule,
          template< typename... > class Action,
          typename Input,
          typename... States >
auto match( Input& in, States&&... st )
  -> std::enable_if_t< has_apply0_v< Rule, Action, States... >, bool >
{
   if( Rule::match< Action >( in, st... ) ) {
      Action< Rule >::apply0( st... );
      return true;
   }
   return false;
}
```

```cpp
template< typename Rule,
          template< typename... > class Action,
          typename Input,
          typename... States >
auto match( Input& in, States&&... st )
  -> std::enable_if_t< !has_apply_v< Rule, Action, Input, States... > &&
                       !has_apply0_v< Rule, Action, States... >, bool >
{
   return Rule::match< Action >( in, st... );
}
```

```
struct a : one< 'a' > {};
struct b : one< 'b' > {};
struct c : one< 'c' > {};
struct ab : seq< a, b > {};
struct ac : seq< a, c > {};
struct g : sor< ab, ac > {};

template< typename Rule > struct action {};
template<> struct action< a > { static void apply0() { std::cout << 'a'; } };
template<> struct action< b > { static void apply0() { std::cout << 'b'; } };
template<> struct action< c > { static void apply0() { std::cout << 'c'; } };
template<> struct action< g > { static void apply0() { std::cout << 'g'; } };

int main()
{
   memory_input in( "ac", "" );
   parse< grammar, action >( in );  // prints aacg
}
```

The Simpletronik

- Generic layer between grammar rules.
- Supports actions and states.
- Calls `apply0()` or `apply()` if applicable.
- Separates the actions (and states) from the grammar.



pegtl::parse< Rule, Action >(in, st…);

Rule::match< Action >(in, st…);

The Simpletronik

pegtl::match< Rule, Action >(in, st…);

Action< Rule >::apply0(st…);
(optional)

Action< Rule >::apply(ai, st…);
(optional)

The Duseltronik

- General purpose customisation points.
  - Trace a parsing run.
  - Generate a parse tree.
  - More *control* over a parsing run.
- How? Another level of indirection.

```
// The Simpletronik
template< typename Rule,
          template< typename... > class Action,
          typename Input,
          typename... States >
auto match( Input& in, States&&... st )
  -> std::enable_if_t< has_apply_v< Rule, Action, Input, States... >, bool >
{
   const auto begin = in.current();
   if( Rule::match< Action >( in, st... ) ) {
      const auto end = in.current();
      const action_input< Input > ai( begin, end );
      Action< Rule >::apply( ai, st... );
      return true;
   }
   return false;
}
```

```
// The Duseltronik
template< typename Rule,
          template< typename... > class Action,
          template< typename... > class Control,
          typename Input,
          typename... States >
auto match( Input& in, States&&... st )
  -> std::enable_if_t< has_apply_v< Rule, Action, Control, Input, States... >, bool >
{
   Control< Rule >::start( in, st... );
   const auto begin = in.current();
   if( Rule::match< Action, Control >( in, st... ) ) {
      Control< Rule >::apply( begin, in, st... );
      Control< Rule >::success( in, st... );
      return true;
   }
   Control< Rule >::failure( in, st... );
   return false;
}
```

```
template< typename... Rules >
struct sor
{
   template< template< typename... > class Action,
             template< typename... > class Control,
             typename Input,
             typename... States >
   static bool match( Input& in, States&&... st )
   {
      return ( pegtl::match< Rules, Action, Control >( in, st... ) || ... );
   }
};
```

```
template< typename... Rules >
struct sor
{
   template< template< typename... > class Action,
             template< typename... > class Control,
             typename Input,
             typename... States >
   static bool match( Input& in, States&&... st )
   {
      return ( Control< Rules >::match< Action, Control >( in, st... ) || ... );
   }
};
```

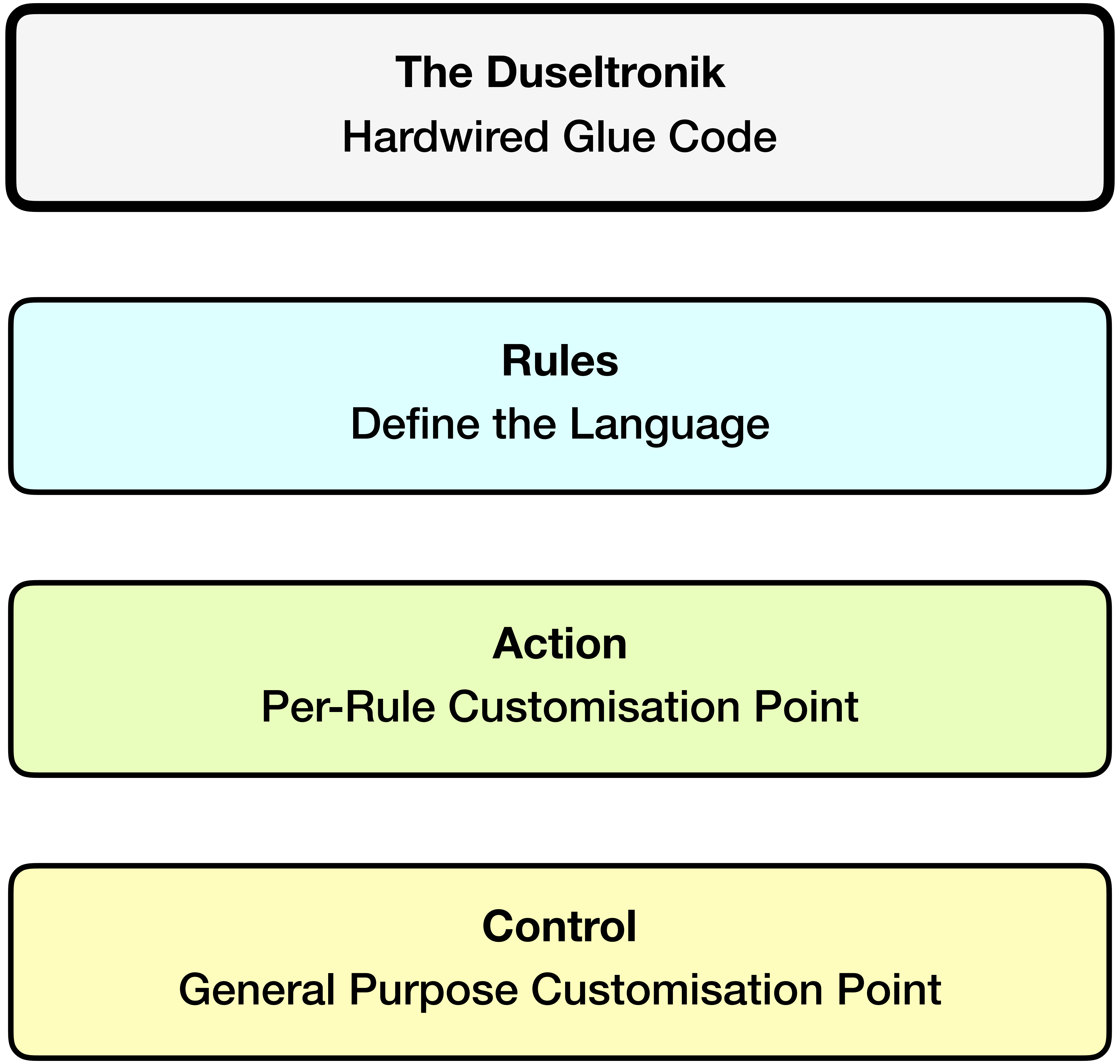
```
template< typename Rule >
struct normal  // default control class template
{
   template… static bool match( Input& in, States&&... st );

   template… static void start( const Input& in, States&&... st ) {}
   template… static void success( const Input& in, States&&... st ) {}
   template… static void failure( const Input& in, States&&... st ) {}

   template…
   static auto apply( const Iterator& begin, const Input& in, States&&... st )
     -> …;

   template…
   static auto apply0( const Input& in, States&&... st )
     -> …;
};
```

```cpp
template< typename Rule >
struct normal
{
   template< template< typename... > class Action,
             template< typename... > class Control,
             typename Input,
             typename... States >
   static bool match( Input& in, States&&... st )
   {
      return pegtl::match< Rule, Action, Control >( in, st... );
   }
};
```

```
template< typename Rule >
struct normal
{
   template< template< typename... > class Action,
             template< typename... > class Control,
             typename Input,
             typename... States >
   static bool match( Input& in, States&&... st )
   {
      if constexpr( has_match_v< Rule, Action, Control, Input, States... > ) {
         return Action< Rule >::match< Rule, Action, Control >( in, st... );
      }
      else {
         return pegtl::match< Rule, Action, Control >( in, st... );
      }
   }
};
```

```
template< typename Rule >
struct normal
{
   template< template< typename... > class Action,
             typename Iterator,
             typename Input,
             typename... States,
             typename AI = typename Input::action_t >
   static auto apply( const Iterator& begin, const Input& in, States&&... st )
     -> decltype( Action< Rule >::apply( std::declval< const AI& >(), st... ) )
   {
      const AI ai( begin, in );
      return Action< Rule >::apply( ai, st... );
   }
};
```

```
template< typename Rule >
struct normal
{
   template< template< typename... > class Action,
             typename Input,
             typename... States >
   static auto apply0( const Input& in, States&&... st )
     -> decltype( Action< Rule >::apply0( st... ) )
   {
      return Action< Rule >::apply0( st... );
   }
};
```

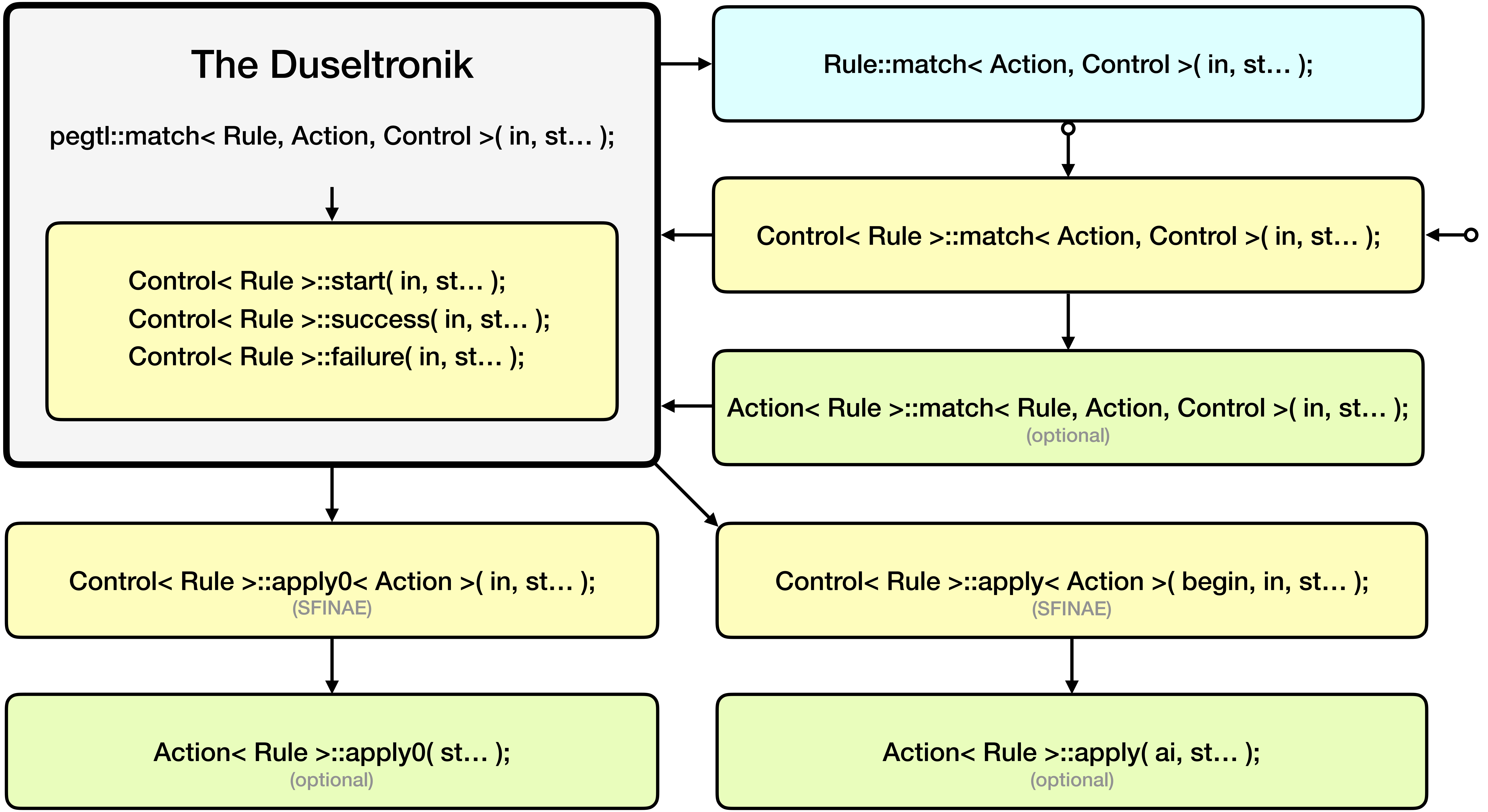


The Duseltronik
Hardwired Glue Code

Rules
Define the Language

Action
Per-Rule Customisation Point

Control
General Purpose Customisation Point



The Duseltronik

pegtl::match< Rule, Action, Control >(in, st…);

Control< Rule >::start(in, st…);
Control< Rule >::success(in, st…);
Control< Rule >::failure(in, st…);

Rule::match< Action, Control >(in, st…);

Control< Rule >::match< Action, Control >(in, st…);

Action< Rule >::match< Rule, Action, Control >(in, st…);
(optional)

Control< Rule >::apply0< Action >(in, st…);
(SFINAE)

Control< Rule >::apply< Action >(begin, in, st…);
(SFINAE)

Action< Rule >::apply0(st…);
(optional)

Action< Rule >::apply(ai, st…);
(optional)

Tracer

```
template< typename Rule >
struct tracer
   : normal< Rule >
{
   template< typename Input, typename... States >
   static void start( const Input& in, States&&... st )
   {
      // add current position, etc. as needed
      std::cerr << "start " << demangle< Rule >() << '\n';
   }

   // likewise for success()/failure()
```

…continued on next slide…

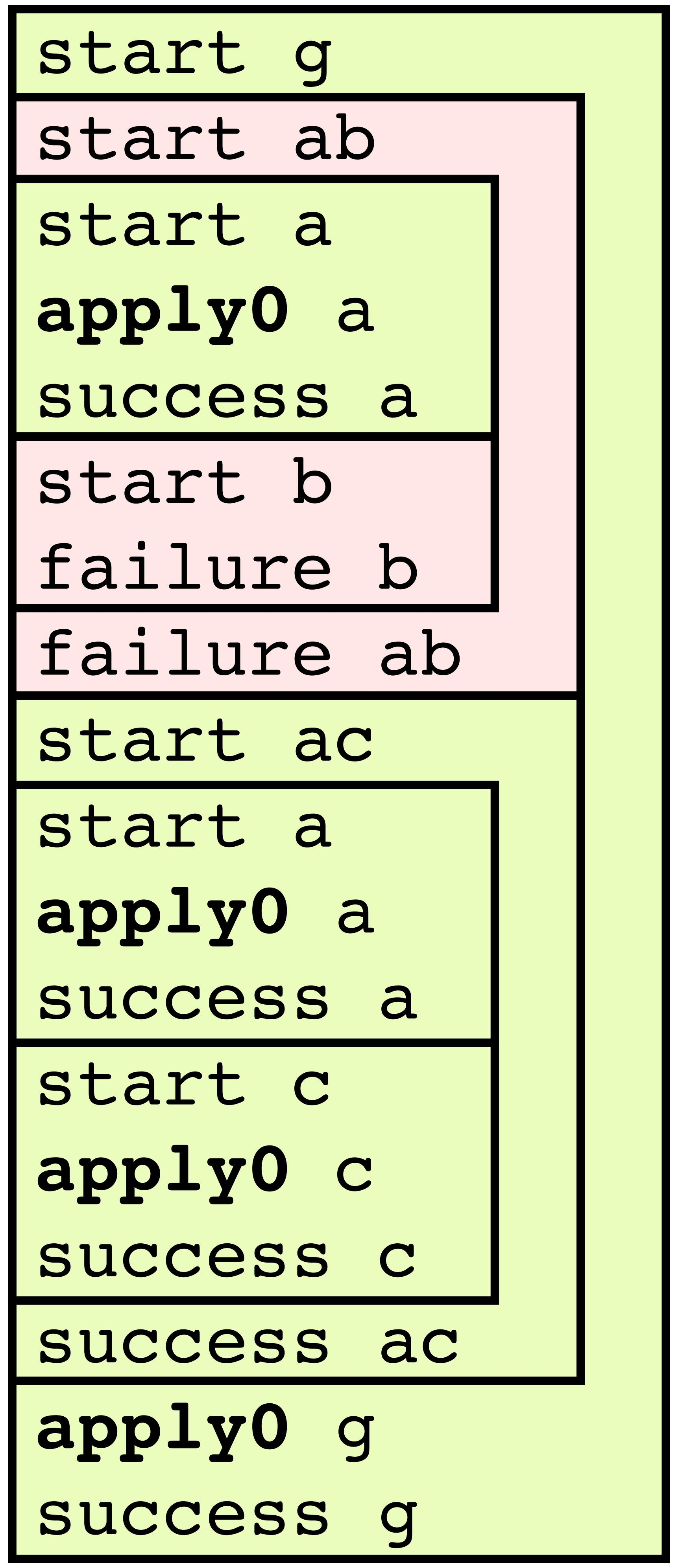
```
   …continued from previous slide…

   template< template< typename... > class Action,
             typename Input,
             typename... States >
   static auto apply0( const Input& in, States&&... st )
     -> decltype( normal< Rule >::apply0< Action >( in, st... ) )
   {
      // add current position, etc. as needed
      std::cerr << "apply0 " << demangle< Rule >() << '\n';
      return normal< Rule >::apply0< Action >( in, st... );
   }

   // likewise for apply()
};
```

```
struct a : one< 'a' > {};
struct b : one< 'b' > {};
struct c : one< 'c' > {};
struct ab : seq< a, b > {};
struct ac : seq< a, c > {};
struct g : sor< ab, ac > {};

template< typename Rule > struct action {};
template<> struct action< a > { static void apply0() { std::cout << 'a'; } };
template<> struct action< b > { static void apply0() { std::cout << 'b'; } };
template<> struct action< c > { static void apply0() { std::cout << 'c'; } };
template<> struct action< g > { static void apply0() { std::cout << 'g'; } };

int main()
{
   memory_input in( "ac", "" );
   parse< grammar, action, tracer >( in );  // prints aacg
}
```

```
struct a : one< 'a' > {};
struct b : one< 'b' > {};
struct c : one< 'c' > {};
struct ab : seq< a, b > {};
struct ac : seq< a, c > {};
struct g : sor< ab, ac > {};

template< typename Rule > struct action {};
template<> struct action< a > { … };
template<> struct action< b > { … };
template<> struct action< c > { … };
template<> struct action< g > { … };

int main()
{
   memory_input in( "ac", "" );
   parse< grammar, action, tracer >( in );
}
```

```
stdout:
aacg
```

stderr:



```
start g
start ab
start a
apply0 a
success a
start b
failure b
failure ab
start ac
start a
apply0 a
success a
start c
apply0 c
success c
success ac
apply0 g
success g
```

Partial Trace

```
template< template< typename... > class NewControl >
struct change_control
{
   template< typename Rule,
             template< typename... > class Action,
             template< typename... > class Control,
             typename Input,
             typename... States >
   static bool match( Input& in, States&&... st )
   {
      return pegtl::match< Rule, Action, NewControl >( in, st... );
   }
};
```

```
template< typename Rule > struct action {};

template<> struct action< my_rule > : change_control< tracer > {};
```

Parse Tree

```
namespace tao::pegtl::parse_tree
{
   struct node
   {
      std::type_index id = typeid( void );
      internal::iterator_t begin, end;
      std::vector< std::unique_ptr< node > > children;
   };

   using state = std::vector< std::unique_ptr< node > >;

   template< typename Rule >
   struct control : normal< Rule >
   {
      template… static void start( const Input& in, state& st );
      template… static void success( const Input& in, state& st );
      template… static void failure( const Input& in, state& st );
   };
}
```

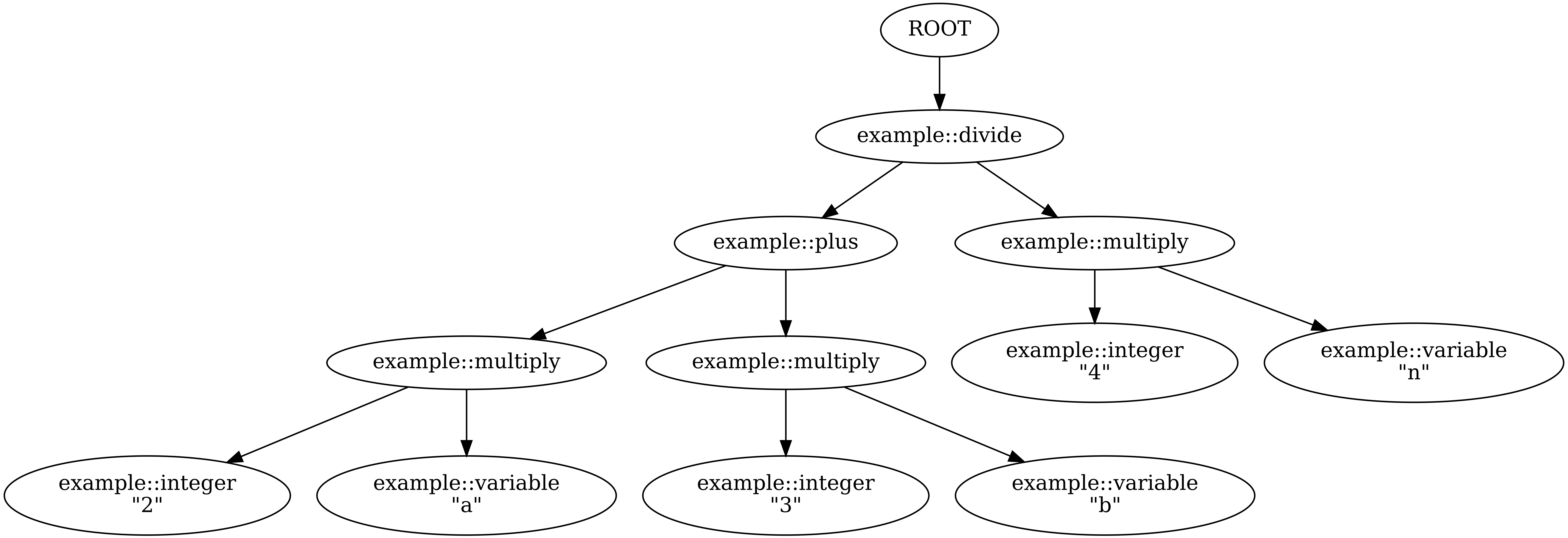
```
namespace tao::pegtl::parse_tree
{
   template< typename Rule >
   template< typename Input >
   void control< Rule >::start( const Input& in, state& st )
   {
      auto n = std::make_unique< node >();
      n->id = typeid( Rule );
      n->begin = in.current();
      st.emplace_back( std::move( n ) );
   }
}
```

```
namespace tao::pegtl::parse_tree
{
   template< typename Rule >
   template< typename Input >
   void control< Rule >::success( const Input& in, state& st )
   {
      auto n = std::move( st.back() );
      st.pop_back();
      n->end = in.current();
      st.back()->children.emplace_back( std::move( n ) );
   }

   template< typename Rule >
   template< typename Input >
   void control< Rule >::failure( const Input& in, state& st )
   {
      st.pop_back();
   }
}
```

```
namespace tao::pegtl::parse_tree
{
   template< typename Rule, typename Input >
   std::unique_ptr< node > parse( const Input& in )
   {
      state st;
      st.emplace_back( std::make_unique< node >() );
      if( pegtl::parse< Rule, pegtl::nothing, control >( in ) ) {
         return std::move( st.back() );
      }
      return nullptr;
   }
}
```

```
int main( int argc, char** argv )
{
   assert( argc == 2 );
   argv_input in( argv, 1 );
   if( const auto root = parse_tree::parse< example::grammar, … >( in ) ) {
      parse_tree::print_dot( std::cout, *root );
      return 0;
   }
   return 1;
}
```

```
> ./parse_tree "(2*a + 3*b) / (4*n)" | dot -Tsvg -o parse_tree.svg
```



Generated Code

[live demo]

Thank You!

https://github.com/taocpp/PEGTL

Questions?

https://github.com/taocpp/PEGTL