AST Construction with the Universal Tree

Bryce Lelbach, LSU
blelbach@cct.lsu.edu
wash, #lll on OFTC, ##spirit on Freenode

github.com/brycelelbach

Overview

- Static → Dynamic
- What's a compiler?
- Semantic actions are evil
- Use the attribute grammar, Luke!
- Meet utree
- LISP interpreter

Note: All snippets are pseudocode, in namespace boost::spirit::qi, forward declarations omitted for clarity

Static → Dynamic

```
rule<Iter, Space> factor =
    uint_
    |   '(' >> expr >> ')'
    |   ('-' >> factor)
    |   ('+' >> factor)
    ;

rule<Iter, Space> term =
    factor
    >> *(   ('*' >> factor)
        |   ('/' >> factor)
        )
    ;

rule<Iter, Space> expr =
    term
    >> *(   ('+' >> term)
        |   ('-' >> term)
        )
    ;
```

Static → Dynamic

```
rule<Iter, Space> factor =
    uint_
    |   '(' >> expr >> ')'
    |   ('-' >> factor)
    |   ('+' >> factor)
    ;

rule<Iter, Space> term =
    factor
    >> *(   ('*' >> factor)
        |   ('/' >> factor)
        )
    ;

rule<Iter, Space> expr =
    term
    >> *(   ('+' >> term)
        |   ('-' >> term)
        )
    ;
```

```
(define factor
  (qi:|
    (qi:uint_)
    (qi:>> (qi:char_ "(") (expr) (qi:char_ ")"))
    (qi:>> (qi:char_ "-") (factor))
    (qi:>> (qi:char_ "+") (factor)))
    )

(define term
  (qi:>> (factor)
      (qi:*
          (qi:|
              (qi:>> (qi:char_ "*") (factor))
              (qi:>> (qi:char_ "/") (factor)))))
              )

(define expr
  (qi:>> (term)
      (qi:*
          (qi:|
              (qi:>> (qi:char_ "+") (term))
              (qi:>> (qi:char_ "-") (term)))))
              )
```

What's a Compiler?

- Parsing
  - Syntax validation
- Semantics
  - Program validation
  - Give source code meaning
  - High level, language dependent optimizations (some compilers)
  - Instruction selection and scheduling

- Parse tree
  - No representation of semantics
- Abstract Syntax Tree
  - Built from the parse tree
  - Represents the behavior of the source code

What's a Compiler?

- Optimization / Code Gen
  - Language dependent
    - Some CFG and DCE optimizations
    - Sibling and tail calls
  - Higher-level language independent
    - Inlining
    - Low-level CFG passes
    - Loop optimizations
  - Lower-level machine independent
    - Some SSA passes
  - Machine dependent (mostly SSA-based)
    - Register allocation
    - Instruction selection and scheduling
- Link Time Optimization
  - Whole-program passes

- Low level IR hierarchy
  - Derived from the AST
  - Control Flow Graph
  - Tree forms
  - SSA forms
  - RTL forms
  - Bytecode
- Object Code

What's a Compiler?

- GCC IR hierarchy
  - GENERIC (tree language, language independent)
  - GIMPLE (SSA language, refinement of GENERIC)
  - RTL (register transfer language, very low-level)
- Clang/LLVM IR hierarchy
  - Clang AST (language dependent)
  - LLVM Assembly (SSA language, language independent)
- PathScale
  - WHIRL languages
  - 5 separate forms

Semantic Actions are Evil

```
rule<Iter, int(), Space> expr =
    term                                [_val = _1]
    >> *(   ('+' >> term                [_val += _1])
        |   ('-' >> term                [_val -= _1])
        )
    ;

rule<Iter, int(), Space> term =
    factor                              [_val = _1]
    >> *(   ('*' >> factor              [_val *= _1])
        |   ('/' >> factor              [_val /= _1])
        )
    ;

rule<Iter, int(), Space> factor =
    uint_                               [_val = _1]
    |   '(' >> expr                     [_val = _1] >> ')'
    |   ('-' >> factor                  [_val = -_1])
    |   ('+' >> factor                  [_val = _1])
    ;
```

I want you to never write grammars that look like this!

- **0.) It is harder to read.**
- **1.) It pollutes the attribute grammar.**
- **2.) Our primitives are not first class citizens (rules).**

Semantic Actions are Evil

- Rationale:
  - Syntactically simple grammars are easier to work with.
    - Easier for others to maintain.
    - Easier to debug.
  - I'm an attribute grammar purist.
    - I want all mutations of the Spirit attribute grammar to be explicit.

Semantic Actions are Evil

```
rule<Iter, std::vector<int>(), Space> expr =
    term
    >> *(   ('+' >> term)
        |   ('-' >> term)
        )
    ;

rule<Iter, std::vector<int>(), Space> term =
    factor
    >> *(   ('*' >> factor)
        |   ('/' >> factor)
        )
    ;

rule<Iter, std::vector<int>(), Space> factor =
    uint_
    |   '(' >> expr >> ')'
    |   ('-' >> factor)
    |   ('+' >> factor)
    ;
```

If we remove the semantic actions, the grammar is much easier to read, and the attribute grammar has not been implicitly hacked. However, we loose all of our semantic information!

Semantic Actions are Evil

```
rule<Iter, ???, Space> factor    = uint_ |'(' >> expr >> ')' | minus_ | plus_;

rule<Iter, ???, Space> term      = factor >> *(times_ | divides_);

rule<Iter, ???, Space> expr      = term >> *(positive_ | negative_);

rule<Iter, ???, Space> minus_    = '-' >> factor;

rule<Iter, ???, Space> plus_     = '+' >> factor;

rule<Iter, ???, Space> times_    = '*' >> factor;

rule<Iter, ???, Space> divides_  = '/' >> factor;

rule<Iter, ???, Space> negative_ = '+' >> term;

rule<Iter, ???, Space> positive_ = '-' >> term;
```

Now, let's divide atoms of semantic meta-data into separate rules. This is closer to what I want to see! But, what attributes do we associate with each rule?

Use the Attribute Grammar, Luke!

- Essential concept: The Spirit attribute grammar can be changed on an application-by-application basis through the use of Spirit-style customization points (SSCPs).
- This means we can gather semantic information and implement tree building by modifying the attribute grammar.

Use the Attribute Grammar, Luke!

- Example of a simple SSCP:

```
template <typename T, /* more template parameters */, typename Enable = void>
struct nifty_hook {
  typedef some_type type;

  static type call (T const&) { /* ... */ }

  /* call overloads */
};

template <typename T>
typename nifty_hook<T>::type nifty (T const& t)
{ return nifty_hook<T>::call(t); }
```

Use the Attribute Grammar, Luke!

- How to associate actions with a rule by modifying the attribute grammar:
  - Determine the actual (RHS) attribute of the rule.
  - Create a type that can hold both the actual attribute data, as well as the semantic information you need to store.
  - Specialize the appropriate SSCPs in Spirit to handle your type.
    - In most cases, this means specializing transform_attribute<>
  - Specify the type as the rule's synthesized attribute.

Use the Attribute Grammar, Luke!

```
namespace client
{
    template <typename A, typename B>
    struct data
    {
        A a;
        B b;
    };

    template <typename Iterator, typename A, typename B>
    struct data_grammar : grammar<Iterator, data<A, B>()>
    {
        data_grammar() : data_grammar::base_type(start)
        {
            start = real_start;
            real_start = auto_ >> ',' >> auto_;
        }

        qi::rule<Iterator, data<A, B>()> start;
        qi::rule<Iterator, fusion::vector<A&, B&>()> real_start;
    };
}
```

Use the Attribute Grammar, Luke!

```
namespace boost { namespace spirit { namespace traits
{
    template <typename A, typename B>
    struct transform_attribute<client::data<A, B>, fusion::vector<A&, B&>, qi::domain>
    {
        typedef fusion::vector<A&, B&> type;

        static type pre(client::data<A, B>& val) { return type(val.a, val.b); }
        static void post(client::data<A, B>&, fusion::vector<A&, B&> const&) {}
        static void fail(client::data<A, B>&) {}
    };
}}}
```

Use the Attribute Grammar, Luke!

```
template <>
struct transform_attribute<utree::nil_type, unused_type, karma::domain> {
  typedef unused_type type;

  static unused_type pre (utree::nil_type&)
  { return unused_type(); }
};
```

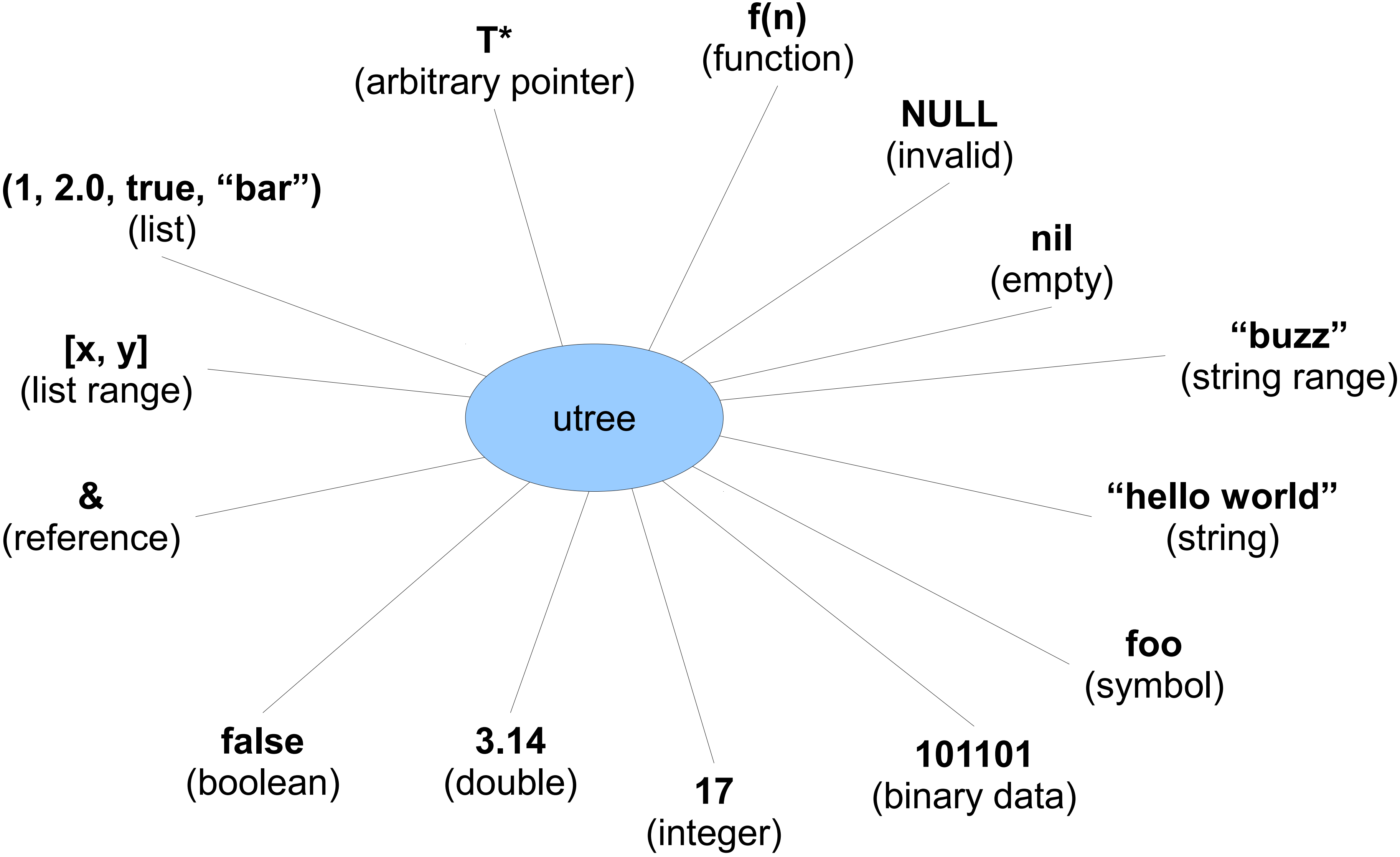
Meet utree

- Need for utree:
  - A generic, hierarchical tree structure is needed for the attribute grammar specialization design strategy. Additionally, some of the more obscure SSCPs are not documented and slightly painful to work with. This makes integrating a tree structure with Spirit through attribute grammar specialization a bit painful.
    - This encourages the use of semantic actions.

Meet utree

- Requirements for utree:
  - Limited or no use of virtual functions and runtime type information.
  - Minimal memory footprint.
  - Minimal dependencies (even Boost and STL).
  - Dynamic type system.
  - Mechanisms for user extension.

Meet utree



utree

T*
(arbitrary pointer)

f(n)
(function)

NULL
(invalid)

nil
(empty)

“buzz”
(string range)

“hello world”
(string)

foo
(symbol)

101101
(binary data)

17
(integer)

3.14
(double)

false
(boolean)

&
(reference)

[x, y]
(list range)

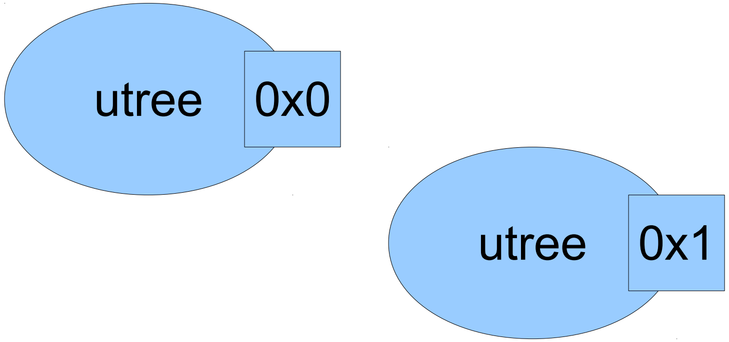
(1, 2.0, true, “bar”)
(list)

Meet utree

- Implementation details:
    - Discriminated union used for type punning.
    - Lists are doubly linked.
    - sizeof(void*[4]).
    - STL interface
    - Small strings are stored directly, large strings are allocated on the heap.
    - Supports Boost.Variant style visitation.

Meet utree

0x0: <node information>
0x1: <node information>
...
...
...
0xn: <node information>



- Tags (integral data, up to ¾ * sizeof(void*) bytes) can be stored in utree nodes.
- These tags can be used to store indexes into an annotation table.
- This allows the association of user-specified meta-data with utree nodes
  - For example, source location references (e.g. line, column and file info)

Meet utree

```
rule<Iter, utree(), Space> factor    = uint_ |'(' >> expr >> ')' | minus_ | plus_;

rule<Iter, utree(), Space> term      = factor >> *(times_ | divides_);

rule<Iter, utree(), Space> expr      = term >> *(positive_ | negative_);

rule<Iter, utree(), Space> minus_    = char_('-') >> factor;

rule<Iter, utree(), Space> plus_     = char_('+') >> factor;

rule<Iter, utree(), Space> times_    = char_('*') >> factor;

rule<Iter, utree(), Space> divides_  = char_('/') >> factor;

rule<Iter, utree(), Space> negative_ = char_('+') >> term;

rule<Iter, utree(), Space> positive_ = char_('-') >> term;
```

Meet utree

```
rule<Iter, utree(), Space> factor    = uint_ |'(' >> expr >> ')' | minus_ | plus_;

rule<Iter, utree(), Space> term      = factor >> *(times_ | divides_);

rule<Iter, utree(), Space> expr      = term >> *(positive_ | negative_);

rule<Iter, utree(), Space> minus_    = char_('-') >> factor;

rule<Iter, utree(), Space> plus_     = char_('+') >> factor;

rule<Iter, utree(), Space> times_    = char_('*') >> factor;

rule<Iter, utree(), Space> divides_  = char_('/') >> factor;

rule<Iter, utree(), Space> negative_ = char_('+') >> term;

rule<Iter, utree(), Space> positive_ = char_('-') >> term;
```

The operators are problematical, because utrees can represent both strings and symbols. Creating a utree from a char (which is the attribute of char_) creates a utree string node.

Meet utree

```
rule<Iter, utf8_symbol_type()> fact_sym = char_('+-');
rule<Iter, utree(), Space> fact_op      = fact_sym >> factor;
rule<Iter, utree(), Space> factor       = uint_ |'(' >> expr >> ')' | fact_op;

rule<Iter, utf8_symbol_type()> term_sym = char_('*/');
rule<Iter, utree(), Space> term_op      = term_sym >> factor;
rule<Iter, utree(), Space> term         = factor >> *term_op;

rule<Iter, utf8_symbol_type()> expr_sym = char_('+-');
rule<Iter, utree(), Space> expr_op      = expr_sym >> term;
rule<Iter, utree(), Space> expr         = term >> *expr_op;
```

Meet utree

- What's utf8_symbol_type?
  - Essentially an alias for std::string, except in the eyes of Spirit's attribute grammar.
  - The attribute grammar specializations create a utree symbol node instead of a utree string under the hood.

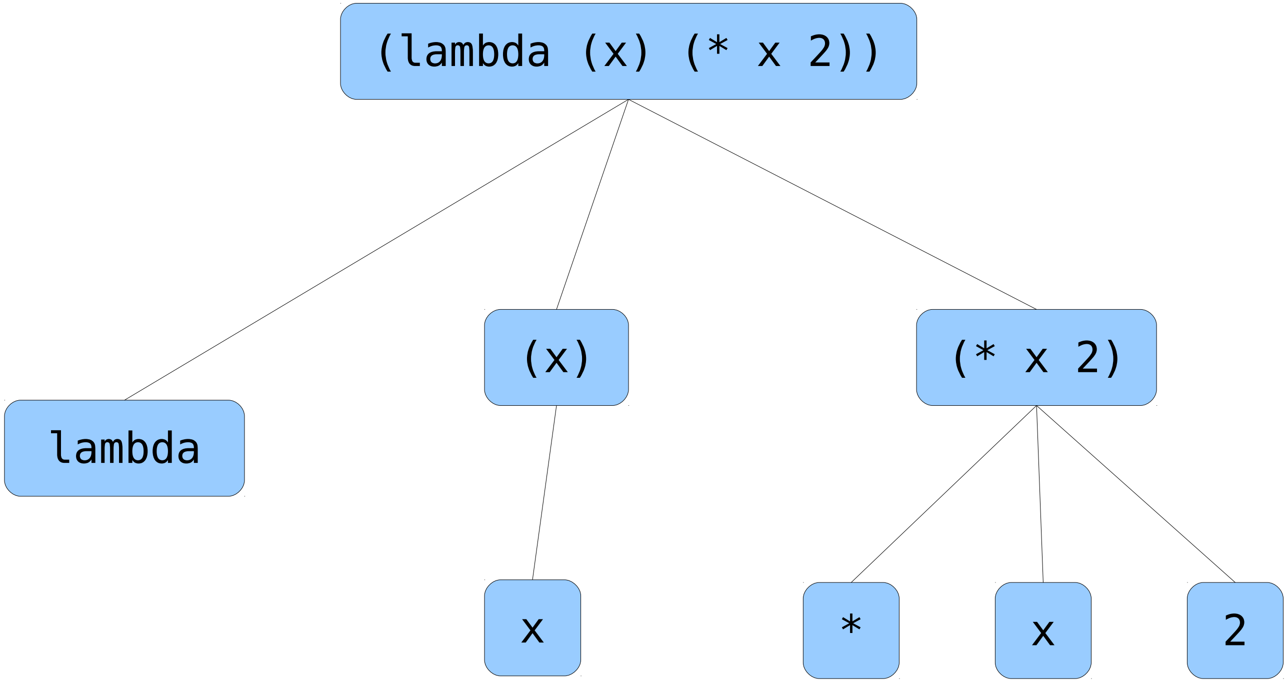
LISP Interpreter

- Phoenix expressions (phxpr, Spirit devs also refer to this as Dynamic Phoenix).
  - Building block for Dynamic Spirit and Dynamic Proto.
  - Draws from Scheme, Python and C++.
  - Minimal design.
- We have a more diverse language specification, but today, I'll just be covering a small subset.

LISP Interpreter

- Lambda expressions
  - `(lambda <formals> <body>)`
- Variable references
  - `<variable>`
- Procedure calls
  - `(<operator> <operand1> …)`
- `Four primitive procedures:`
  - `+, -, * and /`

LISP Interpreter

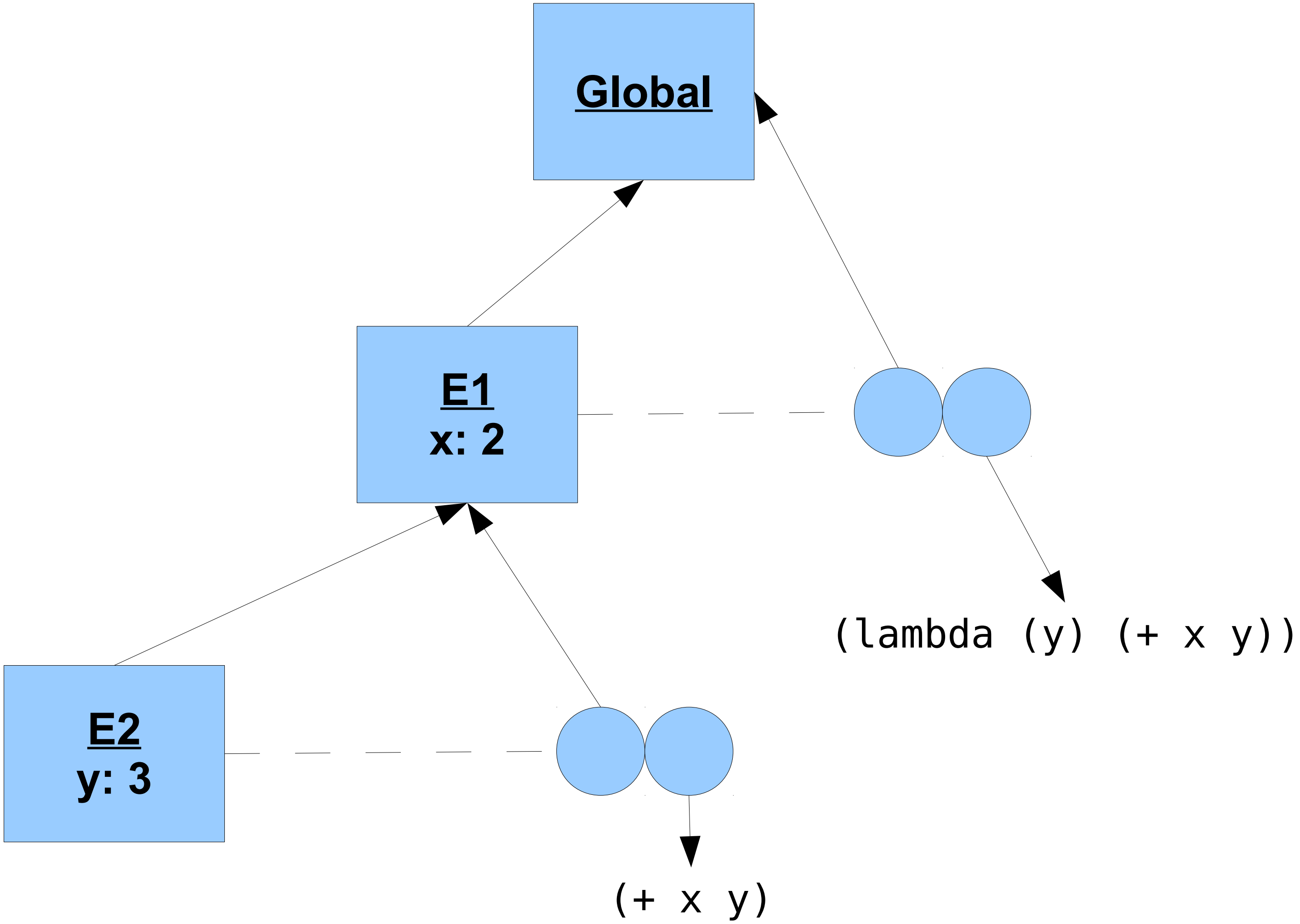


LISP Interpreter

- A lambda expression returns a procedure.
- Procedures are closures.
  - Function + referenced environment = procedure
  - A procedure binds the free variables of its function. This makes procedures first class objects.
- A procedure call consists of:
  - Extend the referenced environment by binding local variables (formals) to their corresponding arguments.

LISP Interpreter

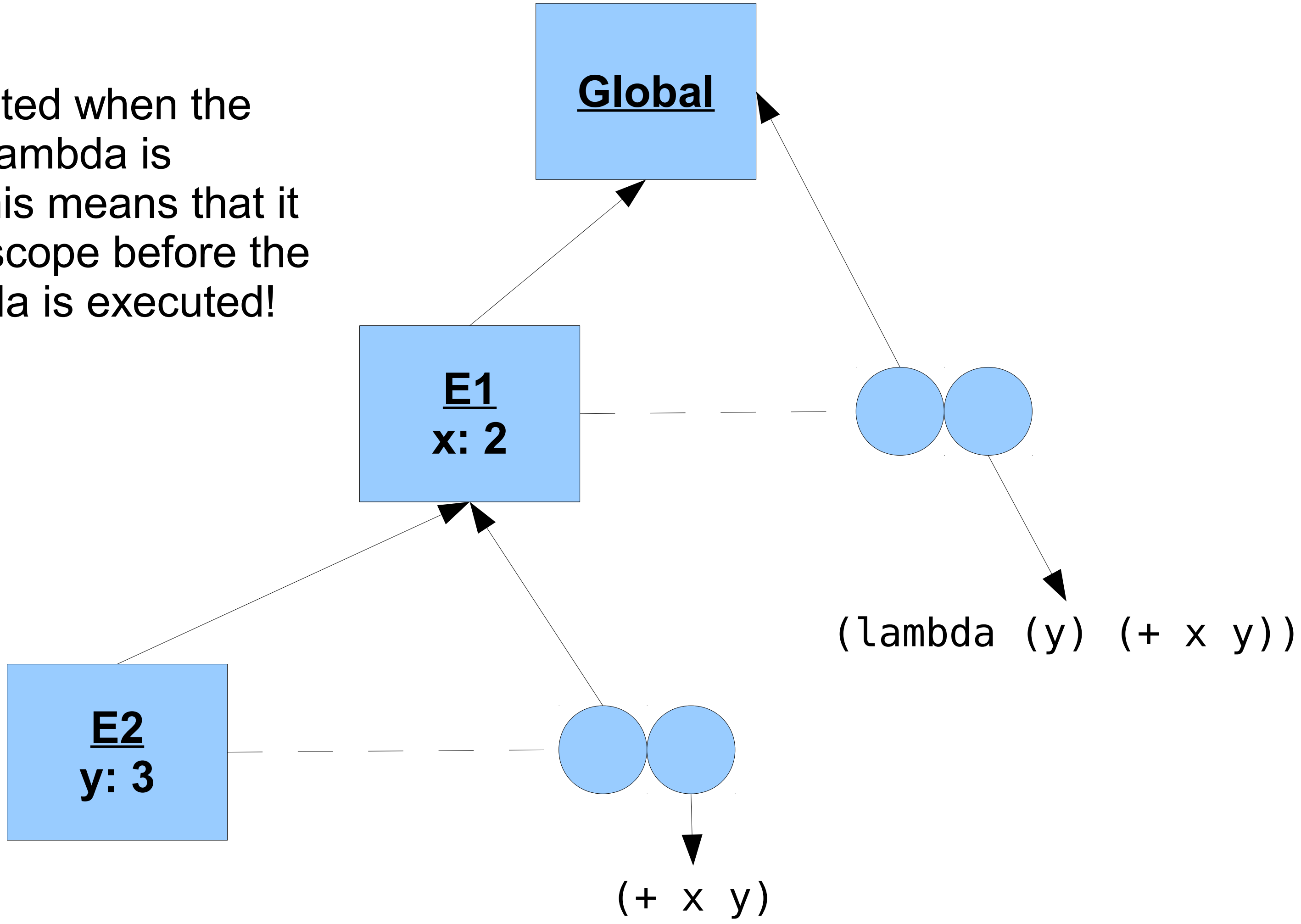
`(((lambda (x) (lambda (y) (+ x y))) 2) 3)`



LISP Interpreter

```
(((lambda (x) (lambda (y) (+ x y))) 2) 3)
```

E1 is allocated when the outermost lambda is invoked. This means that it will go out scope before the inner lambda is executed!



Global

E1
x: 2

E2
y: 3

`(lambda (y) (+ x y))`

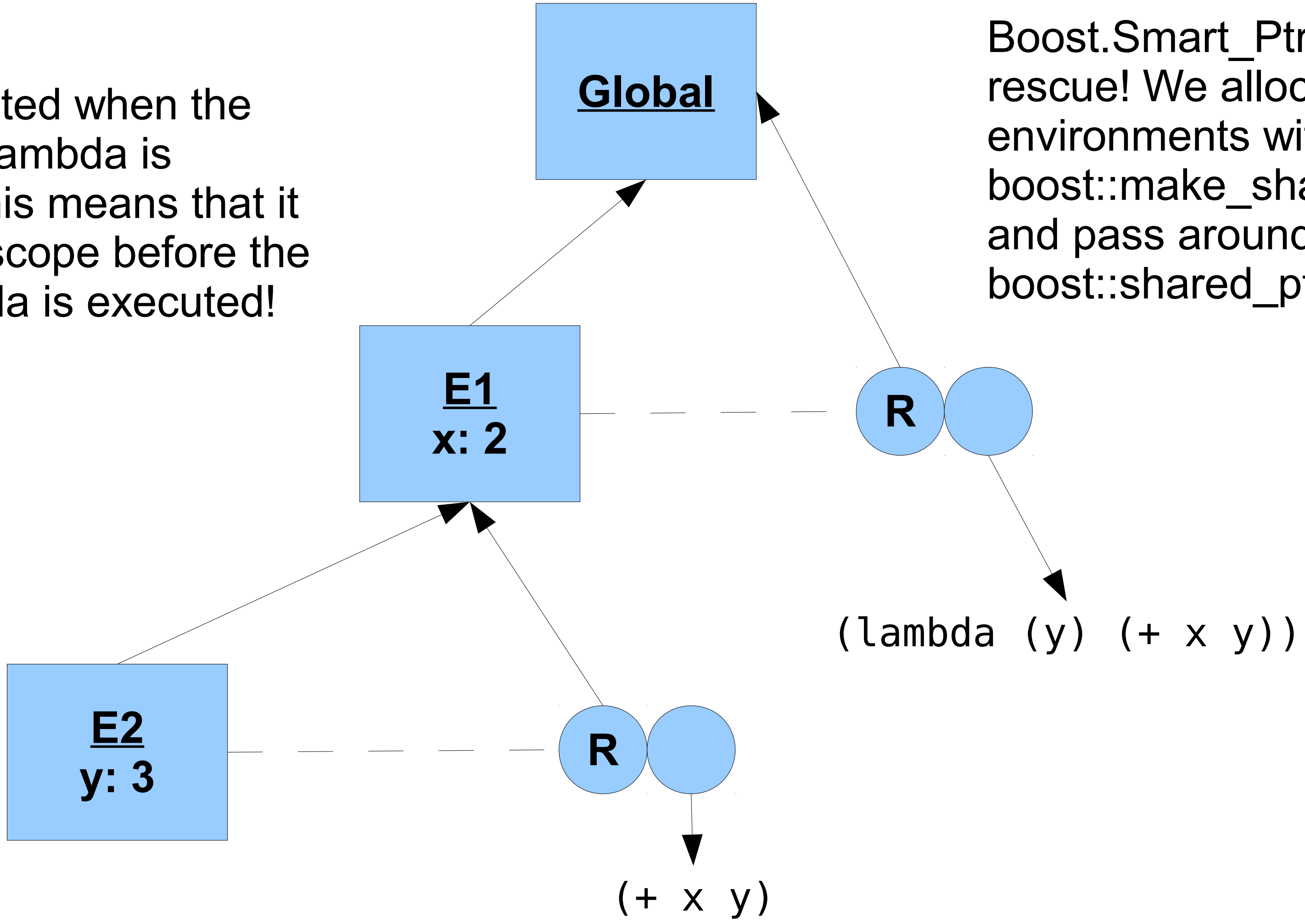
`(+ x y)`

LISP Interpreter

`(((lambda (x) (lambda (y) (+ x y))) 2) 3)`

E1 is allocated when the outermost lambda is invoked. This means that it will go out scope before the inner lambda is executed!

Boost.Smart_Ptr to the rescue! We allocate all environments with boost::make_shared<>, and pass around boost::shared_ptr<>s.



Global

E1
x: 2

R

`(lambda (y) (+ x y))`

E2
y: 3

R

`(+ x y)`

LISP Interpreter

- Boost.Smart_Ptr is an essential building block of a utree/Spirit based compiler.
  - Not only useful for implementing garbage-collecting languages!
  - Reference counting can be used to perform simple dead store elimination.

LISP Interpreter

Algorithms + Data Structures = Programs

LISP Interpreter

- Data structures:
  - Signature
  - Environment (compile-time)
  - Scope (run-time)
  - Evaluator
    - Global procedure table
  - Function objects
    - Function body
    - Procedure
    - Lambda
    - Placeholder
    - Thunk

LISP Interpreter

```
struct arity_type {
  enum info { fixed, variable };
};

struct function_type {
  enum info { placeholder, /* others */ };
};

typedef std::size_t displacement

typedef fusion::vector3<displacement, arity_type::info, function_type::info> signature;

struct environment {
  boost::unordered_map<utree, boost::shared_ptr<utree> > definitions;
  boost::shared_ptr<environment> parent;
  boost::weak_ptr<utree> this_;
};

struct scope: boost::enable_shared_from_this<scope> {
  boost::shared_array<utree> elements;
  boost::shared_ptr<scope> parent;
  const displacement level;
};
```

LISP Interpreter

```
// acts like std::vector
typedef sheol::adt::dynamic_array<signature> global_procedure_table;

struct evaluator {
  boost::shared_ptr<environment> variables;
  boost::shared_ptr<global_procedure_table> gpt;
  const displacement frame;
};

typedef sheol::adt::dynamic_array<utree> code;

struct function_body {
  boost::shared_ptr<code_type> code_;
};

struct procedure {
  boost::shared_ptr<function_body> body;
  boost::shared_ptr<scope> parent_env;
  const signature sig;
};
```

LISP Interpreter

```
struct lambda {
  boost::shared_ptr<function_body> body;
  const signature sig;
};

struct placeholder {
  const displacement n;
  const displacement frame;
};

struct thunk {
  boost::shared_ptr<code_type> code_;
  boost::shared_ptr<global_procedure_table> gpt;
};
```

LISP Interpreter

- Algorithms:
  - Compile a lambda expression
    - Make placeholders
    - Compile a lambda body
  - Evaluate a variable reference
  - Evaluate a procedure call
    - Evaluate a function
  - Evaluate a compiled lambda
    - Evaluate a placeholder
    - Evaluate a thunk
    - Make lazy calls

LISP Interpreter

```
utree evaluator::evaluate_lambda_expression (utree const& formals, utree const& body) {
  evaluator local_env(variables, gpt, frame + 1);

  make_placeholders(formals, local_env);

  boost::shared_ptr<function_body> fbody = boost::make_shared<function_body>();

  utree::const_iterator it = body.begin(), end = body.end();

  for (; it != end; ++it) {
    utree f = evaluate_lambda_body(*it, local_env);
    fbody->code->push_back(f);
  }

  const signature sig(formals.size(), arity_type::fixed /* more metadata */);
  lambda l(fbody, sig);

  local_env.gpt->push_back(sig);
  utree ut = stored_function<lambda>(l);
  ut.tag(local_env.gpt->size() - 1);

  return ut;
}
```

LISP Interpreter

```
void evaluator::make_placeholders (utree const& formals) {
  utree::const_iterator it = formals.begin(), end = formals.end();

  for (utree::size_type i = 0, end = formals.size(); i != end; ++i, ++it) {
    boost::shared_ptr<utree> p = variables->define(*it, utree
      (stored_function<placeholder>(placeholder(i, frame))));

    const signature sig(i, arity_type::fixed /* more metadata */);
    gpt->push_back(sig);
    p->tag(gpt->size() - 1);
  }
}
```

LISP Interpreter

```
utree evaluator::evaluate_lambda_body (utree const& body) {
  boost::shared_ptr<code_type> lazy_call = boost::make_shared<code_type>();

  if (prana::is_utree_container(body)) {
    utree::const_iterator it = body.begin(), end = body.end();

    if ((it != end) && (*it == utree(spirit::utf8_symbol_type("lambda")))) {
      iterator formals = it; ++formals;
      iterator body = formals; ++body;
      lazy_call->push_back
         (evaluate_lambda_expression(*formals, evaluator::range_type(body, end), *this));
    }

    else
      BOOST_FOREACH(utree const& element, body) { lazy_call->push_back(evaluate(element, *this));
}
  }

  else
    lazy_call->push_back(utree::visit(body, *this));

  thunk t(lazy_call, gpt);
  return utree(stored_function<thunk>(t));
}
```

LISP Interpreter

```
utree evaluator::evaluate_variable_reference (utree const& sym) {
  boost::shared_ptr<utree> p = variables->lookup(sym);

  if (!p)
    return utree();

  return *p;
}
```

LISP Interpreter

```
utree procedure::eval (scope const& args) const {
  /* arity checks omitted */

  if (args.size() != 0) {
    boost::shared_array<utree> const& ap = args.checkout();

    boost::shared_ptr<scope> new_scope = boost::make_shared<scope>
      (ap, args.elements->size(), parent_env);
    return body->eval(*new_scope);
  }

  // nullary
  else {
    boost::shared_ptr<scope> new_scope = boost::make_shared<scope>(parent_env);
    return body->eval(*new_scope);
  }
}
```

LISP Interpreter

```
utree function_body::eval (scope const& env) const {
  code_type::size_type i = 0;
  const code_type::size_type end = code->size();

  for (; i != (end - 1); ++i) {
    if (prana::recursive_which((*code)[i]) == utree_type::function_type)
      (*code)[i].eval(env);
  }

  if (prana::recursive_which((*code)[end - 1]) == utree_type::function_type)
    return (*code)[end - 1].eval(env);
  else
    return utree(boost::ref((*code)[end - 1]));
}
```

LISP Interpreter

```
utree lambda::eval (scope const& env) const {
  boost::shared_ptr<scope> saved_env;

  if (env.level() == 0)
    saved_env = env.get();
  else
    saved_env = env.parent.get();

  function_base* pf = new stored_function<procedure>
    (procedure(body, saved_env, sig));

  return utree(pf);
}
```

LISP Interpreter

```
utree placeholder::eval (scope const& env) const {
  boost::shared_ptr<scope> eptr = env.get();

  while (frame != eptr->level())
    eptr = eptr->outer();

  return utree((*eptr)[n]);
}
```

LISP Interpreter

```
utree thunk::eval (scope const& args) const {
  utree const& lazy_f = eval_lazy_call((*lazy_call)[0], args);

  const displacement lazy_env_size = lazy_call->size() - 1;
  boost::shared_array<utree> lazy_env(new utree[lazy_env_size]);

  for (std::size_t i = 0, end = lazy_env_size; i != end; ++i)
    lazy_env[i] = eval_lazy_call((*lazy_call)[i + 1], args);

  if (prana::recursive_which(lazy_f) != utree_type::function_type)
    return lazy_f;

  boost::shared_ptr<scope> new_scope
    = boost::make_shared<scope>(lazy_env, lazy_env_size, args.get());

  return lazy_f.eval(*new_scope);
}
```

LISP Interpreter

```
utree thunk::execute_lazy (utree const& lazy_arg, scope const& args) const {
  using boost::fusion::at_c;

  if (prana::recursive_which(lazy_arg) == utree_type::function_type) {
    BOOST_ASSERT(lazy_arg.tag() <= global_procedure_table->size());

    // Load the lazy argument's signature from the gpt.
    signature const& sig = (*global_procedure_table)[lazy_arg.tag()];

    if (at_c<2>(sig) == function_type::placeholder)
      return lazy_arg.eval(args);
    else
      return utree(boost::ref(lazy_arg));
  }

  else
    return utree(boost::ref(lazy_arg));
}
```

LISP Interpreter

```
utree evaluate (utree const& ut) {
  evaluator ev;
  return evaluate(ut, ev);
}

utree evaluate (utree const& ut, evaluator& ev) {
  return utree::visit(ut, prana::visit_ref(ev));
}
```

github.com/brycelelbach/prana