Abstract Syntax Trees

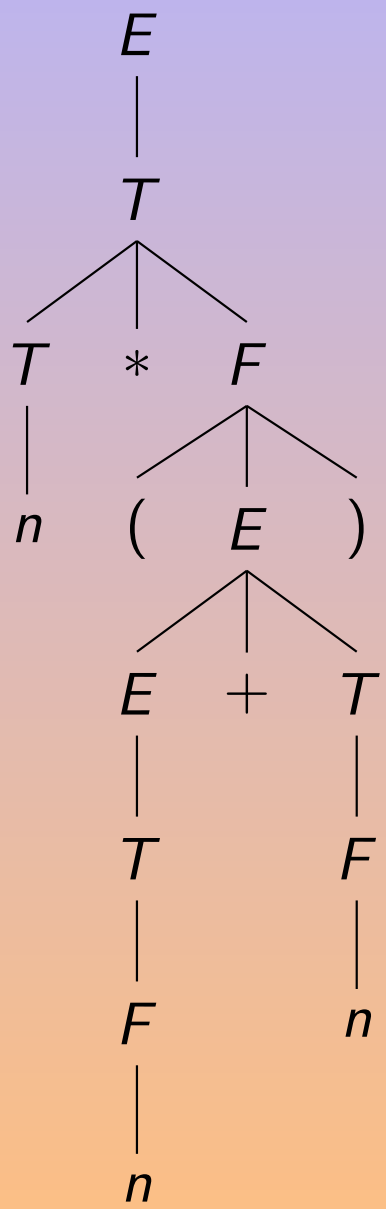
Akim Demaille Étienne Renault Roland Levillain
first.last@lrde.epita.fr

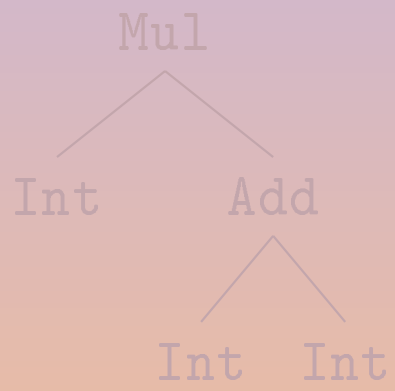
EPITA — École Pour l'Informatique et les Techniques Avancées

February 2, 2020

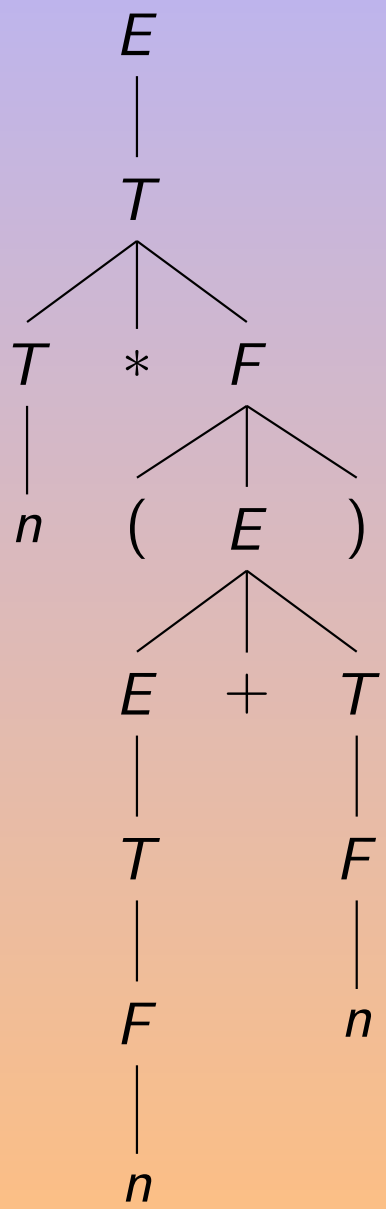
Abstract Syntax Trees

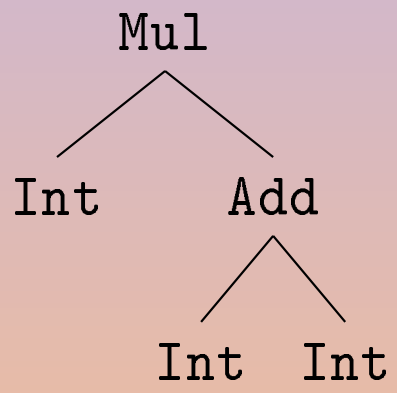
$1 * (2 + 3)$, twice





$1 * (2 + 3)$, twice





Abstract Syntax Tree

- Parse Tree, Concrete Syntax
- Abstract Syntax Tree, Abstract Syntax
- Syntactic Sugar
- Traversals

Abstract Syntax Tree

- Parse Tree, Concrete Syntax
- Abstract Syntax Tree, Abstract Syntax
- Syntactic Sugar
- Traversals

Abstract Syntax Tree

- Parse Tree, Concrete Syntax
- Abstract Syntax Tree, Abstract Syntax
- Syntactic Sugar
- Traversals

Abstract Syntax Tree

- Parse Tree, Concrete Syntax
- Abstract Syntax Tree, Abstract Syntax
- Syntactic Sugar
- Traversals

Structured Data for Input/Output: Trees

AST Generators

Syntax Definition Formalism [Visser, 1995]

```
module Tiger-Expressions
imports Tiger-Lexicals Tiger-Literals
exports
  sorts Exp Var
  context-free syntax
    Id                                  →  Var        {cons("Var")}
    Var                                 →  LValue
    LValue "." Id                       →  LValue     {cons("FieldVar")}
    LValue "[" Exp "]"                  →  LValue     {cons("Subscript")}
    IntConst                            →  Exp        {cons("Int")}
    StrConst                            →  Exp        {cons("String")}
    "nil"                               →  Exp        {cons("NilExp")}
    LValue                              →  Exp
    Var "(" {Exp ","}* ")"              →  Exp        {cons("Call")}
    Id "=" Exp                          →  InitField  {cons("InitField")}
    TypeId "{" {InitField ","}* "}"     →  Exp  {cons("Record")}
    TypeId "[" {Exp ","}+ "]" "of" Exp  →  Exp  {cons("Array")}
```

Exchanging Trees

Abstract Syntax Notation number One

ASN.1 [ASN.1 Consortium, 2003, Dubuisson, 2003]

- an international standard
- specify data used in communication protocols
- powerful and complex language
- describe accurately and efficiently communications between homogeneous or heterogeneous systems

Abstract Syntax Notation number One

ASN.1 [ASN.1 Consortium, 2003, Dubuisson, 2003]

- an international standard
- specify data used in communication protocols
- powerful and complex language
- describe accurately and efficiently communications between homogeneous or heterogeneous systems

Abstract Syntax Notation number One

ASN.1 [ASN.1 Consortium, 2003, Dubuisson, 2003]

- an international standard
- specify data used in communication protocols
- powerful and complex language
- describe accurately and efficiently communications between homogeneous or heterogeneous systems

Abstract Syntax Notation number One

ASN.1 [ASN.1 Consortium, 2003, Dubuisson, 2003]

- an international standard
- specify data used in communication protocols
- powerful and complex language
- describe accurately and efficiently communications between homogeneous or heterogeneous systems

ASN.1

```
Example DEFINITIONS ::=
BEGIN
    AddressType ::= SEQUENCE {
        name            OCTET STRING,
        number          INTEGER,
        street          OCTET STRING,
        apartNumber     INTEGER OPTIONAL,
        postOffice      OCTET STRING,
        state           OCTET STRING,
        zipCode         INTEGER
    }
END
```

ASN.1

Tags to avoid problems similar to matching a*a*.

```
Example DEFINITIONS ::=
BEGIN
    Letter ::= SEQUENCE {
        opening          OCTET STRING,
        body             OCTET STRING,
        closing          OCTET STRING,
        receiverAddr     [0] AddressType OPTIONAL,
        senderAddr       [1] AddressType OPTIONAL
    }
END
```

Protobuff

- Extensible mechanism for serializing data
- Language neutral
- Plateform neutral
- Compact (3 to 10 times smaller than XML)
- Back and Forward compatibility
- Generate data access classes that are easier to use programmatically
- Support for RPC

Protobuff

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}
```

Protobuff

```
// Serialization
Person person;
person.set_name(``John Doe'');
person.set_id(1234);
person.set_email(``jdoe@example.com'');
fstream output(``myfile'', ios::out | ios::binary);
person.SerializeToOstream(&output);
// De-Serialization
fstream input(``myfile'', ios::in | ios::binary);
Person person;
person.ParseFromIstream(&input);
cout << ``Name: `` << person.name() << endl;
cout << ``E-mail: `` << person.email() << endl;
```

ATerms

Grammar of ATerms

```
t   ::= bt                  -- basic term
      | bt { t }            -- annotated term
bt  ::= C                   -- constant
      | C(t1,...,tn)        -- n-ary constructor
      | (t1,...,tn)         -- n-ary tuple
      | [t1,...,tn]         -- list
      | "ccc"               -- quoted string
      | int                 -- integer
      | real                -- floating point number
      | blob                -- binary large object
```

C is a *constructor* name — an identifier or a quoted string [Centrum voor Wiskunde en Informatica, 2004].

Examples of ATerms

constants	`abc`
numerals	`42`
literals	`"asdf"`
lists	`[], [1, "abc" 2], [1, 2, [3, 4]]`
functions	`f("a"), g(1,[])`
annotations	`f("a") {"remark"}`

Other Frameworks

- SGML/XML
  - YAXX: YAcc eXtension to XML [Yu and D'Hollander, 2003]
- CORBA
- JSON
- YAML
- SDL
- S-expressions (sexps)
  - SXML
- Protobuff
- RMI, RPC, Corba

Simple Implementation of ast in C++

Simple Grammar

Concrete Grammar (BNF)

```
⟨exp⟩    ::= ⟨exp⟩ '+' ⟨term⟩
           | ⟨exp⟩'-' ⟨term⟩
           | ⟨term⟩.
⟨term⟩   ::= ⟨term⟩ '*' ⟨factor⟩
           | ⟨term⟩ '/' ⟨factor⟩
           | ⟨factor⟩.
⟨factor⟩ ::= '(' ⟨exp⟩ ')'
           | ⟨num⟩.
```

Abstract Gramnar (RTG)

```
⟨exp⟩ ::= Add(⟨exp⟩, ⟨exp⟩)
        | Sub(⟨exp⟩, ⟨exp⟩)
        | Mul(⟨exp⟩, ⟨exp⟩)
        | Div(⟨exp⟩, ⟨exp⟩)
        | Num(⟨num⟩).
```

Simple Grammar

Concrete Grammar (BNF)

```
⟨exp⟩    ::= ⟨exp⟩ '+' ⟨term⟩
           | ⟨exp⟩'-' ⟨term⟩
           | ⟨term⟩.
⟨term⟩   ::= ⟨term⟩ '*' ⟨factor⟩
           | ⟨term⟩ '/' ⟨factor⟩
           | ⟨factor⟩.
⟨factor⟩ ::= '(' ⟨exp⟩ ')'
           | ⟨num⟩.
```

Abstract Gramnar (RTG)

```
⟨exp⟩ ::= Add(⟨exp⟩, ⟨exp⟩)
        | Sub(⟨exp⟩, ⟨exp⟩)
        | Mul(⟨exp⟩, ⟨exp⟩)
        | Div(⟨exp⟩, ⟨exp⟩)
        | Num(⟨num⟩).
```

Expressions: Exp

```
class Exp
{
protected:
  Exp() = default;
  Exp(const Exp& rhs) = default;
  Exp& operator=(const Exp& rhs) = default;

public:
  virtual ~Exp();
};
```

Binary Expressions: Bin

```
class Bin : public Exp
{
public:
  Bin(char oper, Exp* lhs, Exp* rhs)
    : Exp(), oper_(oper), lhs_(lhs), rhs_(rhs)
  {}

  ~Bin() override
  { delete lhs_; delete rhs_; }

private:
  char oper_; Exp* lhs_; Exp* rhs_;
};
```

Numbers: Num

```
class Num : public Exp
{
public:
  Num(int val)
    : Exp(), val_(val)
  {}

private:
  int val_;
};
```

Constructing an ast

```
int
main()
{
  Exp* tree = new Bin('+', new Num(42), new Num(51));
  delete tree;
}
```

How to process the AST?

Constructing an ast

```
int
main()
{
  Exp* tree = new Bin('+', new Num(42), new Num(51));
  delete tree;
}
```

How to process the AST?

Algorithms on trees: Traversals

Traversals in Compilers

- pretty printer
- name analysis
- unique identifiers
- desugaring
- type checking
- non local (escaping) variables
- inlining
- high level optimizations
- translation to other intermediate representations
- etc.

Traversals in Compilers

- pretty printer
- name analysis
- unique identifiers
- desugaring
- type checking
- non local (escaping) variables
- inlining
- high level optimizations
- translation to other intermediate representations
- etc.

Traversals in Compilers

- pretty printer
- name analysis
- unique identifiers
- desugaring
- type checking
- non local (escaping) variables
- inlining
- high level optimizations
- translation to other intermediate representations
- etc.

Traversals in Compilers

- pretty printer
- name analysis
- unique identifiers
- desugaring
- type checking
- non local (escaping) variables
- inlining
- high level optimizations
- translation to other intermediate representations
- etc.

Traversals in Compilers

- pretty printer
- name analysis
- unique identifiers
- desugaring
- type checking
- non local (escaping) variables
- inlining
- high level optimizations
- translation to other intermediate representations
- etc.

Traversals in Compilers

- pretty printer
- name analysis
- unique identifiers
- desugaring
- type checking
- non local (escaping) variables
- inlining
- high level optimizations
- translation to other intermediate representations
- etc.

Traversals in Compilers

- pretty printer
- name analysis
- unique identifiers
- desugaring
- type checking
- non local (escaping) variables
- inlining
- high level optimizations
- translation to other intermediate representations
- etc.

Traversals in Compilers

- pretty printer
- name analysis
- unique identifiers
- desugaring
- type checking
- non local (escaping) variables
- inlining
- high level optimizations
- translation to other intermediate representations
- etc.

Traversals in Compilers

- pretty printer
- name analysis
- unique identifiers
- desugaring
- type checking
- non local (escaping) variables
- inlining
- high level optimizations
- translation to other intermediate representations
- etc.

Traversals in Compilers

- pretty printer
- name analysis
- unique identifiers
- desugaring
- type checking
- non local (escaping) variables
- inlining
- high level optimizations
- translation to other intermediate representations
- etc.

Tagging the Abstract Syntax Tree

Some traversals discover information that change the translation:

- an escaping variable must not be stored in a register
- the code for `a < b` depends on the types of `a` and `b`
- `a := print_int(51)` must not produce a real assignment

Annotate some ast nodes.

Tagging the Abstract Syntax Tree

Some traversals discover information that change the translation:

- an escaping variable must not be stored in a register
- the code for `a < b` depends on the types of `a` and `b`
- `a := print_int(51)` must not produce a real assignment

Annotate some ast nodes.

Tagging the Abstract Syntax Tree

Some traversals discover information that change the translation:

- an escaping variable must not be stored in a register
- the code for `a < b` depends on the types of `a` and `b`
- `a := print_int(51)` must not produce a real assignment

Annotate some ast nodes.

Tagging the Abstract Syntax Tree

Some traversals discover information that change the translation:

- an escaping variable must not be stored in a register
- the code for `a < b` depends on the types of `a` and `b`
- `a := print_int(51)` must not produce a real assignment

Annotate some ast nodes.

Supporting the operator<<

Expressions: Exp

```
#include <iostream>

class Exp
{
protected:
  Exp() {};
  Exp(const Exp& rhs) {};
  Exp& operator=(const Exp& rhs) {};

public:
  virtual ~Exp() {};
};

std::ostream&
operator<<(std::ostream& o, const Exp& tree)
{
  return o << "Uh oh...";
}
```

Binary Expressions: Bin

```
class Bin : public Exp {
public:
  Bin(char oper, Exp* lhs, Exp* rhs)
    : Exp(), oper_(oper), lhs_(lhs), rhs_(rhs)
  {}
  ~Bin() override { delete lhs_; delete rhs_; }

  friend std::ostream&
  operator<<(std::ostream& o, const Bin& tree);

private:
  char oper_; Exp* lhs_; Exp* rhs_;
};

std::ostream& operator<<(std::ostream& o, const Bin& tree) {
  return o << '(' << *tree.lhs()
           << tree.oper() << *tree.rhs() << ')';
}
```

Numbers: Num

```
class Num : public Exp
{
public:
  Num(int val)
    : Exp(), val_(val)
  {}

  friend std::ostream&
  operator<<(std::ostream& o, const Num& tree);

private:
  int val_;
};

std::ostream&
operator<<(std::ostream& o, const Num& tree)
{
  return o << tree.val_;
}
```

Invoking and Printing

```
int
main()
{
  Bin* bin = new Bin('+', new Num(42), new Num(51));
  Exp* exp = bin;
  std::cout << "Exp: " << *exp << std::endl;
  std::cout << "Bin: " << *bin << std::endl;
  delete bin;
}
```

Using operator<<

```
% ./bin2
Exp: Uh oh...
Bin: (Uh oh...+Uh oh...)
```

- compile time selection (*static binding*)
  based on the containing/variable type.
- We need it at run time (*dynamic binding*)
  based on the contained/object type.
  - also called *inclusion polymorphism*
  - provided by `virtual` in C++

Using operator<<

```
% ./bin2
Exp: Uh oh...
Bin: (Uh oh...+Uh oh...)
```

- compile time selection (*static binding*) based on the containing/variable type.
- We need it at run time (*dynamic binding*) based on the contained/object type.
  - also called *inclusion polymorphism*
  - provided by `virtual` in C++

Using operator<<

```
% ./bin2
Exp: Uh oh...
Bin: (Uh oh...+Uh oh...)
```

- compile time selection (*static binding*)
  based on the containing/variable type.
- We need it at run time (*dynamic binding*)
  based on the contained/object type.
  - also called *inclusion polymorphism*
  - provided by `virtual` in C++

Using operator<<

```
% ./bin2
Exp: Uh oh...
Bin: (Uh oh...+Uh oh...)
```

- compile time selection (*static binding*)
  based on the containing/variable type.
- We need it at run time (*dynamic binding*)
  based on the contained/object type.
  - also called *inclusion polymorphism*
  - provided by `virtual` in C++

Using operator<<

```
% ./bin2
Exp: Uh oh...
Bin: (Uh oh...+Uh oh...)
```

- compile time selection (*static binding*)
  based on the containing/variable type.
- We need it at run time (*dynamic binding*)
  based on the contained/object type.
  - also called *inclusion polymorphism*
  - provided by `virtual` in C++

Expressions: Exp

```
#include <iostream>

class Exp
{
public:
  virtual std::ostream& print(std::ostream& o) const = 0;
};
```

Binary Expressions: Bin

```
class Bin : public Exp
{
public:
  Bin(char op, Exp* l, Exp* r)
    : Exp(), oper_(op), lhs_(l), rhs_(r)
  {}

  ~Bin() override {
    delete lhs_; delete rhs_;
  }

  std::ostream& print(std::ostream& o) const override {
    o << '('; lhs_->print(o); o << oper_;
    rhs_->print(o); return o << ')';
  }

private:
  char oper_; Exp* lhs_; Exp* rhs_;
};
```

Numbers: Num

```
class Num : public Exp
{
public:
  Num(int val) : Exp(), val_(val)
  {}

  std::ostream&
  print(std::ostream& o) const override
  {
    return o << val_;
  }

private:
  int val_;
};
```

Using this ast

```
std::ostream&
operator<<(std::ostream& o, const Exp& e)
{
  return e.print(o);
}

int
main()
{
  Bin* bin = new Bin('+', new Num(42), new Num(51));
  Exp* exp = bin;
  std::cout << "Exp: " << *exp << std::endl;
  std::cout << "Bin: " << *bin << std::endl;
  delete bin;
}
```

Discussion

It works...

```
% ./exp3
Exp: (42+51)
Bin: (42+51)
```

but `Bin::print` is obfuscated.

```
std::ostream&
Bin::print(std::ostream& o) const
{
  o << '(';
  lhs()->print(o);
  o << oper_;
  rhs()->print(o);
  o << ')';
  return o;
}
```

Discussion

It works...

```
% ./exp3
Exp: (42+51)
Bin: (42+51)
```

but `Bin::print` is obfuscated.

```
std::ostream&
Bin::print(std::ostream& o) const
{
  o << '(';
  lhs()->print(o);
  o << oper_;
  rhs()->print(o);
  o << ')';
  return o;
}
```

Making operator<< Polymorphic

Just use the operator<< in print!

```
class Exp {
public:
  virtual std::ostream& print(std::ostream& o) const = 0;
};

std::ostream& operator<<(std::ostream& o, const Exp& e) {
  return e.print(o);
}

std::ostream& Bin::print(std::ostream& o) const {
  return o << '(' << *lhs() << oper() << *rhs() << ')';
}
```

Cuter, but you cannot pass additional arguments to print.

Making operator<< Polymorphic

Just use the operator<< in print!

```
class Exp {
public:
  virtual std::ostream& print(std::ostream& o) const = 0;
};

std::ostream& operator<<(std::ostream& o, const Exp& e) {
  return e.print(o);
}

std::ostream& Bin::print(std::ostream& o) const {
  return o << '(' << *lhs() << oper() << *rhs() << ')';
}
```

Cuter, but you cannot pass additional arguments to print.

Separate processing and dispatching

- In the previous code, `operator<<` processes and dispatches
- Additional operations will require processing and dispatching

Processing

- Keep it external
- Add new easily

Dispatching

- Keep it internal
- Once for all: Factor it!

Separate processing and dispatching

- In the previous code, `operator<<` processes and dispatches
- Additional operations will require processing and dispatching

Processing

- Keep it external
- Add new easily

Dispatching

- Keep it internal
- Once for all: Factor it!

Separate processing and dispatching

- In the previous code, `operator<<` processes and dispatches
- Additional operations will require processing and dispatching

Processing

- Keep it external
- Add new easily

Dispatching

- Keep it internal
- Once for all: Factor it!

Separate processing and dispatching

- In the previous code, `operator<<` processes and dispatches
- Additional operations will require processing and dispatching

Processing

- Keep it external
- Add new easily

Dispatching

- Keep it internal
- Once for all: Factor it!

operator<< to process

```
std::ostream& operator<<(std::ostream& o, const Bin& e)
{
  return o << '(' << *e.lhs() << oper() << *e.rhs() << ')';
}

std::ostream& operator<<(std::ostream& o, const Num& e)
{
  return o << e.val;
}

std::ostream& operator<<(std::ostream& o, const Exp& e)
{
  return e.print(o);
}
```

operator<< to process

```
std::ostream& operator<<(std::ostream& o, const Bin& e)
{
  return o << '(' << *e.lhs() << oper() << *e.rhs() << ')';
}

std::ostream& operator<<(std::ostream& o, const Num& e)
{
  return o << e.val;
}

std::ostream& operator<<(std::ostream& o, const Exp& e)
{
  return e.print(o);
}
```

print to dispatch

```
class Exp {
public:
  virtual std::ostream& print(std::ostream& o) const = 0;
};

class Bin {
public:
  std::ostream& print(std::ostream& o) const override {
    return o << *this;
  }
  ...
};

class Num {
public:
  std::ostream& print(std::ostream& o) const override {
    return o << *this;
  }
  ...
```

Separate processing and dispatching

- Now `operator<<` processes
- `print` dispatches
- Each processing requires its dispatching
- Pass pointers to functions to factor the dispatching?

Separate processing and dispatching

- Now `operator<<` processes
- `print` dispatches
- Each processing requires its dispatching
- Pass pointers to functions to factor the dispatching?

Separate processing and dispatching

- Now `operator<<` processes
- `print` dispatches
- Each processing requires its dispatching
- Pass pointers to functions to factor the dispatching?

Multimethods

Multimethods

- Polymorphism over any argument, not only just on the object:

```
using std::ostream;
ostream& operator<<(ostream& o, virtual const Exp& e);
ostream& operator<<(ostream& o, virtual const Bin& e);
ostream& operator<<(ostream& o, virtual const Num& e);
```

- This is called multimethods
- CLOS, Common Lisp Object System

Multimethods

- Polymorphism over any argument, not only just on the object:

```
using std::ostream;
ostream& operator<<(ostream& o, virtual const Exp& e);
ostream& operator<<(ostream& o, virtual const Bin& e);
ostream& operator<<(ostream& o, virtual const Num& e);
```

- This is called multimethods
- CLOS, Common Lisp Object System

Multimethods

- Polymorphism over any argument, not only just on the object:

```
using std::ostream;
ostream& operator<<(ostream& o, virtual const Exp& e);
ostream& operator<<(ostream& o, virtual const Bin& e);
ostream& operator<<(ostream& o, virtual const Num& e);
```

- This is called multimethods
- CLOS, Common Lisp Object System

Multimethods in C++

- No multimethods in C++03/11/14/17
- Simulate via a *trampoline*

```
std::ostream&
operator<<(std::ostream& o, const Exp& e)
{
  return e.print(o);
}

virtual std::ostream& Exp::print(std::ostream& o) = 0;
        std::ostream& Bin::print(std::ostream& o) override;
        std::ostream& Num::print(std::ostream& o) override;
```

- Ask the hierarchy to perform the dispatch

Multimethods in C++

- No multimethods in C++03/11/14/17
- Simulate via a *trampoline*

```
std::ostream&
operator<<(std::ostream& o, const Exp& e)
{
  return e.print(o);
}

virtual std::ostream& Exp::print(std::ostream& o) = 0;
        std::ostream& Bin::print(std::ostream& o) override;
        std::ostream& Num::print(std::ostream& o) override;
```

- Ask the hierarchy to perform the dispatch

Multimethods in C++

- No multimethods in C++03/11/14/17
- Simulate via a *trampoline*

```
std::ostream&
operator<<(std::ostream& o, const Exp& e)
{
  return e.print(o);
}

virtual std::ostream& Exp::print(std::ostream& o) = 0;
        std::ostream& Bin::print(std::ostream& o) override;
        std::ostream& Num::print(std::ostream& o) override;
```

- Ask the hierarchy to perform the dispatch

Multimethods in C++

- Ask the hierarchy to perform the dispatch
- Additional work on the hierarchy
- The concept is spread in several files
- Requires the ability to edit the hierarchy

Multimethods in C++

- Ask the hierarchy to perform the dispatch
- Additional work on the hierarchy
- The concept is spread in several files
- Requires the ability to edit the hierarchy

Multimethods in C++

- Ask the hierarchy to perform the dispatch
- Additional work on the hierarchy
- The concept is spread in several files
- Requires the ability to edit the hierarchy

Multimethods in C++

- Ask the hierarchy to perform the dispatch
- Additional work on the hierarchy
- The concept is spread in several files
- Requires the ability to edit the hierarchy

Did you get it?

Ask
the hierarchy
to perform
the dispatch!

Ask the Hiearchy to Dispatch

```
class Exp
{
public:
  virtual ~Exp() = default;

  using bin_t = std::function<auto (const Bin&) -> void>;
  using num_t = std::function<auto (const Num&) -> void>;
  virtual void dispatch(bin_t bin, num_t num) const = 0;
};
```

Ask the Hiearchy to Dispatch

```
void Bin::dispatch(bin_t bin, num_t) const
{
  bin(*this);
}

void Num::dispatch(bin_t, num_t num) const
{
  num(*this);
}
```

Ask the Hiearchy to Dispatch

```
std::ostream& operator<<(std::ostream& o, const Bin& b)
{
  return o << *b.lhs() << ' ' << b.oper() << ' ' << *b.rhs();
}

std::ostream& operator<<(std::ostream& o, const Num& n)
{
  return o << n.val();
}

std::ostream& operator<<(std::ostream& o, const Exp& e)
{
  e.dispatch([&o](const Bin& b) { o << b; },
             [&o](const Num& n) { o << n; });
  return o;
}
```

Ask the Hiearchy to Dispatch

```
int main()
{
  Exp* exp = new Bin('+', new Num(42), new Num(51));
  std::cout << *exp << std::endl;
  delete exp;
}
```

Ask the Hiearchy to Dispatch

```
int main()
{
  Exp* exp = new Bin('+', new Num(42), new Num(51));
  std::cout << *exp << std::endl;
  delete exp;
}
42 + 51
```

Dispatch: Classes

- It works!
- But what if we introduce a new class?
- What if how hierarchy has 10 classes?

Dispatch: Classes

- It works!
- But what if we introduce a new class?
- What if how hierarchy has 10 classes?

> *If you have a procedure with ten parameters, you probably missed some.*
>
> 
>
> — Epigrams on Programming, Alan Perlis, 1982

Dispatch: Arguments

- Support for indentation: a new argument is needed.
- Similarly if we want to return a value.
- Introduce structures carried in the traversals.

```
struct stick_t
{
  std::ostream& ostr;
  int res;
  unsigned tab;
};
```

- Better yet: make them objects.

Dispatch: Arguments

- Support for indentation: a new argument is needed.
- Similarly if we want to return a value.
- Introduce structures carried in the traversals.

```
struct stick_t
{
  std::ostream& ostr;
  int res;
  unsigned tab;
};
```

- Better yet: make them objects.

Dispatch: Arguments

- Support for indentation: a new argument is needed.
- Similarly if we want to return a value.
- Introduce structures carried in the traversals.

```
struct stick_t
{
  std::ostream& ostr;
  int res;
  unsigned tab;
};
```

- Better yet: make them objects.

Dispatch: Arguments

- Support for indentation: a new argument is needed.
- Similarly if we want to return a value.
- Introduce structures carried in the traversals.

```
struct stick_t
{
  std::ostream& ostr;
  int res;
  unsigned tab;
};
```

- Better yet: make them objects.

Visitors

Visitors

Visitors encapsulate the traversal data and algorithm.

```
class PrettyPrinter
{
public:
  void visitBin(const Bin& e) {
    ostr_ << '('; ...
  }
  void visitNum(const Num& e); {
    ostr_ << e.val_;
  }

private:
  std::ostream& ostr_;
  unsigned tab_;
};
```

Class Visitor

```
#include <iostream>

// Fwd.
class Exp;
class Bin;
class Num;

class Visitor
{
public:
  virtual void visitBin(const Bin& exp) = 0;
  virtual void visitNum(const Num& exp) = 0;
};
```

Classes Exp and Num

```
class Exp {
public:
  virtual void accept(Visitor& v) const = 0;
};


class Num : public Exp {
public:
  Num(int val)
    : Exp(), val_(val)
  {}

  void accept(Visitor& v) const override {
    v.visitNum(*this);
  }

private:
  int val_;
};
```

Class Bin

```
class Bin : public Exp
{
public:
  Bin(char op, Exp* l, Exp* r)
    : Exp(), oper_(op), lhs_(l), rhs_(r)
  {}

  ~Bin() override {
    delete lhs_; delete rhs_;
  }

  void accept(Visitor& v) const override {
    v.visitBin(*this);
  }

private:
  char oper_; Exp* lhs_; Exp* rhs_;
};
```

Class PrettyPrinter

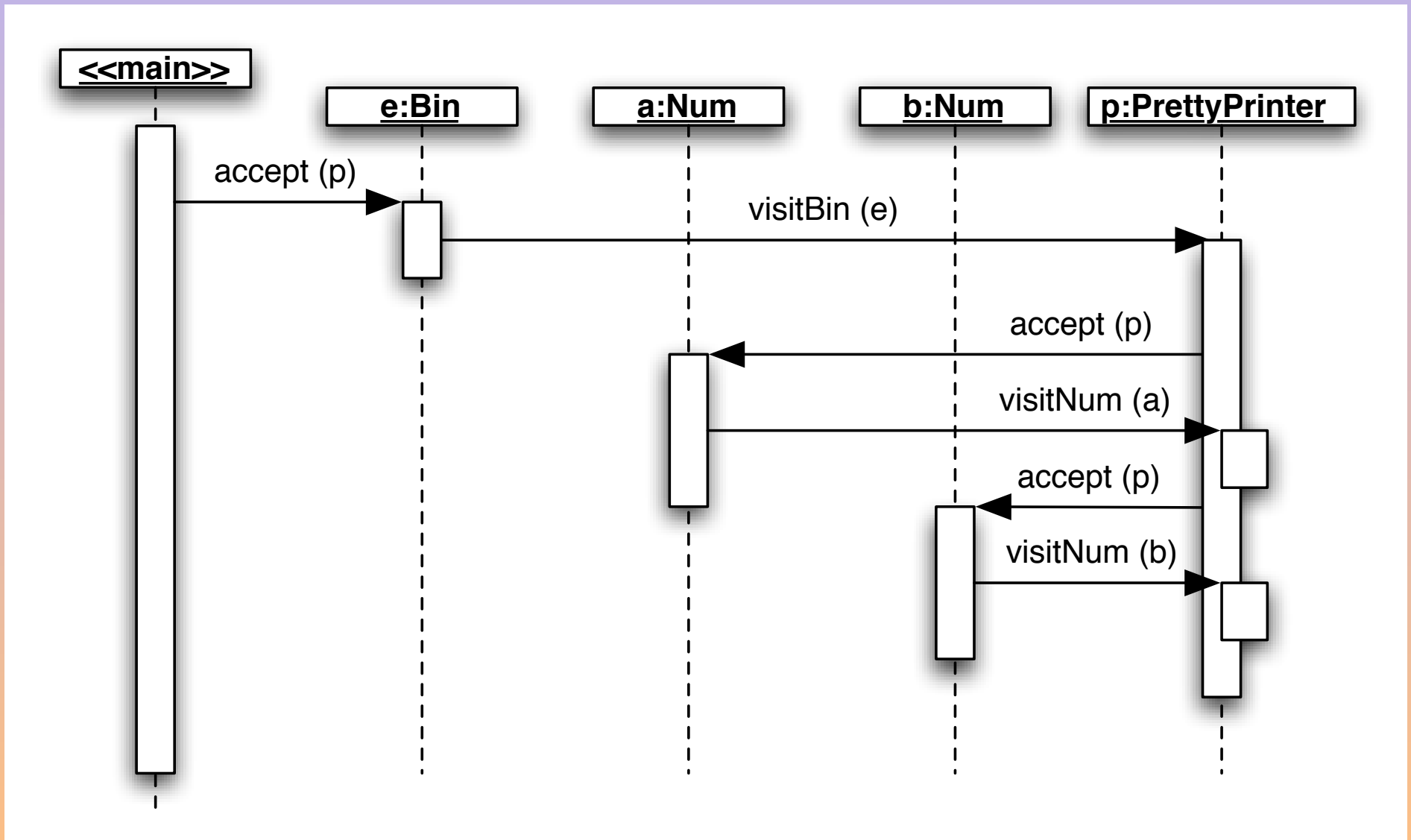
```
class PrettyPrinter : public Visitor
{
public:
  PrettyPrinter(std::ostream& ostr)
    : ostr_(ostr) {}

  void visitBin(const Bin& e) override {
    ostr_ << '('; e.lhs()->accept(*this);
    ostr_ << e.oper(); e.rhs()->accept(*this); ostr_ << ')';
  }

  void visitNum(const Num& e) override {
    ostr_ << e.val_;
  }

private:
  std::ostream& ostr_;
};
```

operator<< and main

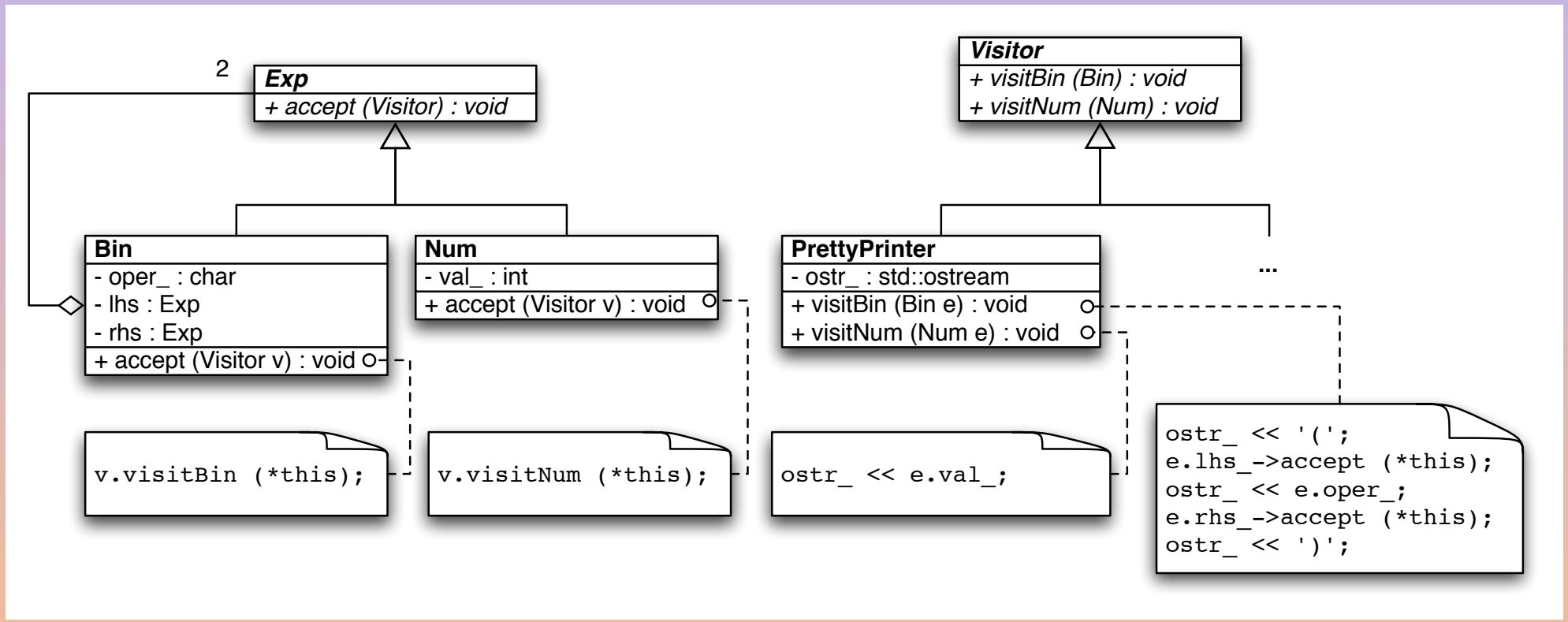
```
std::ostream&
operator<<(std::ostream& o, const Exp& e)
{
  auto printer = PrettyPrinter{o};
  e.accept(printer);
  return o;
}

int
main()
{
  Bin* bin = new Bin('+', new Num(42), new Num(51));
  Exp* exp = bin;
  std::cout << "Bin: " << *bin << std::endl;
  std::cout << "Exp: " << *exp << std::endl;
  delete bin;
}
```

A pretty-printing sequence diagram

```
Exp* a = new Num(42); Exp* b = new Num(51);
Exp* e = new Bin('+', a, b); std::cout << *e << std::endl;
```



A class diagram: Visitor and Composite design patterns



Further with Visitors

Visitors in C++

- `Visitor` and `ConstVisitor`
  similar to `iterator` and `const_iterator`
- Use C++ templates to factor
  (e.g., `Visitor` and `ConstVisitor`, see the lecture on generic programming)
- Use C++ overloading
  only `visit` instead of `visitBin` and `visitNum`

Visitors in C++

- `Visitor` and `ConstVisitor`
  similar to `iterator` and `const_iterator`
- Use C++ templates to factor
  (e.g., `Visitor` and `ConstVisitor`, see the lecture on generic programming)
- Use C++ overloading
  only `visit` instead of `visitBin` and `visitNum`

Visitors in C++

- `Visitor` and `ConstVisitor`
  similar to `iterator` and `const_iterator`
- Use C++ templates to factor
  (e.g., `Visitor` and `ConstVisitor`, see the lecture on generic programming)
- Use C++ overloading
  only `visit` instead of `visitBin` and `visitNum`

Object Functions

- How about `operator()` instead of `visit`?
- Does not help the user, pure for implementation convenience
- But then, we can improve this

```
int eval(const Exp& e) {
  auto eval = Evaluator{};
  return e.evaluate(eval);
}
```

provided

Object Functions

- How about `operator()` instead of `visit`?
- Does not help the user, pure for implementation convenience
- But then, we can improve this

```
int eval(const Exp& e) {
  auto eval = Evaluator{};
  return e.evaluate(eval);
}
```

provided

Object Functions

- How about `operator()` instead of `visit`?
- Does not help the user, pure for implementation convenience
- But then, we can improve this

```
int eval(const Exp& e) {
  auto eval = Evaluator{};
  e.accept(eval);
  return eval.value;
}
```

provided

```
int eval(const Exp& e) {
  auto eval = Evaluator{};
  eval(e);
  return eval.value;
}
```

Object Functions

- How about `operator()` instead of `visit`?
- Does not help the user, pure for implementation convenience
- But then, we can improve this

```
int eval(const Exp& e) {
  auto eval = Evaluator{};
  e.accept(eval);
  return eval.value;
}
```

provided

```
int eval(const Exp& e) {
  auto eval = Evaluator{};
  eval(e);
  return eval.value;
}
```

Object Functions

- How about `operator()` instead of `visit`?
- Does not help the user, pure for implementation convenience
- But then, we can improve this

```
int eval(const Exp& e) {
  auto eval = Evaluator{};
  e.accept(eval);
  return eval.value;
}
```

```
int eval(const Exp& e) {
  auto eval = Evaluator{};
  eval(e);
  return eval.value;
}
```

provided

```
void Evaluator::operator()(const Exp& e) {
  e.accept(*this);
}
```

Sugaring Visitors 1

```
struct Evaluator : public ConstVisitor {
  void operator()(const Exp& e) override { e.accept(*this); }
  void operator()(const Bin& e) override {
    e.lhs()->accept(*this); int lhs = value;
    e.rhs()->accept(*this); int rhs = value;
    ... value = lhs + rhs; ...
  }
  void operator()(const Num& e) override { value = e.val; }
  int value;
};

int eval(const Exp& e) {
  auto eval = Evaluator{};
  eval(e);
  return eval.value;
}
```

Sugaring Visitors 2

```
struct Evaluator : public ConstVisitor
{
  ...
  void
  operator()(const Bin& e) override {
    ...
    value = eval(e.lhs())
          + eval(e.rhs());
    ...
  }

  void
  operator()(const Num& e) override {
    value = e.val;
  }

  int value;
};
```

One visitor per eval invocation

- A useless cost
- Easy automatic variables
- Harder for shared data (no static please!)

Sugaring Visitors 2

```
struct Evaluator : public ConstVisitor
{
  ...
  void
  operator()(const Bin& e) override {
    ...
    value = eval(e.lhs())
          + eval(e.rhs());
    ...
  }

  void
  operator()(const Num& e) override {
    value = e.val;
  }

  int value;
};
```

One visitor per eval invocation

- A useless cost
- Easy automatic variables
- Harder for shared data (no static please!)

Sugaring Visitors 2

```
struct Evaluator : public ConstVisitor
{
  ...
  void
  operator()(const Bin& e) override {
    ...
    value = eval(e.lhs())
          + eval(e.rhs());
    ...
  }

  void
  operator()(const Num& e) override {
    value = e.val;
  }

  int value;
};
```

One visitor per eval invocation

- A useless cost
- Easy automatic variables
- Harder for shared data (no static please!)

Sugaring Visitors 2

```
struct Evaluator : public ConstVisitor
{
  ...
  void
  operator()(const Bin& e) override {
    ...
    value = eval(e.lhs())
          + eval(e.rhs());
    ...
  }

  void
  operator()(const Num& e) override {
    value = e.val;
  }

  int value;
};
```

One visitor per eval invocation

- A useless cost
- Easy automatic variables
- Harder for shared data (no static please!)

Sugaring Visitors 2

```
struct Evaluator : public ConstVisitor
{
  ...
  void
  operator()(const Bin& e) override {
    ...
    value = eval(e.lhs())
          + eval(e.rhs());
    ...
  }

  void
  operator()(const Num& e) override {
    value = e.val;
  }

  int value;
};
```

One visitor per eval invocation

- A useless cost
- Easy automatic variables
- Harder for shared data (no static please!)

Sugaring Visitors 3

```
struct Evaluator : public ConstVisitor
{
  int eval(const Exp& e) {
    e.accept(*this); return value;
  }

  void operator()(const Exp& e) { e.accept(*this); }
  void operator()(const Bin& e) override {
    ...
    value = eval(e.lhs()) + eval(e.rhs());
    ...
  }
  void operator()(const Num& e) override {
    value = e.val;
  }

  int value;
};
```

Sugaring the PrettyPrinter

```
void
PrettyPrinter::operator()(const Bin& e)
  override
{
  ostr_ << '(';
  e.lhs()->accept(*this);
  ostr_ << e.oper();
  e.rhs()->accept(*this);
  ostr_ << ')';
}
```

- We could insert a `print` method
- But that's not nice
- We can use the `operator<<`
- But we no longer can pass additional arguments
- Unless...

Sugaring the PrettyPrinter

```
void
PrettyPrinter::operator()(const Bin& e)
  override
{
  ostr_ << '(';
  e.lhs()->accept(*this);
  ostr_ << e.oper();
  e.rhs()->accept(*this);
  ostr_ << ')';
}
```

- We could insert a `print` method
- But that's not nice
- We can use the `operator<<`
- But we no longer can pass additional arguments
- Unless...

Sugaring the PrettyPrinter

```
void
PrettyPrinter::operator()(const Bin& e)
  override
{
  ostr_ << '(';
  e.lhs()->accept(*this);
  ostr_ << e.oper();
  e.rhs()->accept(*this);
  ostr_ << ')';
}
```

- We could insert a `print` method
- But that's not nice
- We can use the `operator<<`
- But we no longer can pass additional arguments
- Unless...

Sugaring the PrettyPrinter

```
void
PrettyPrinter::operator()(const Bin& e)
  override
{
  ostr_ << '(';
  e.lhs()->accept(*this);
  ostr_ << e.oper();
  e.rhs()->accept(*this);
  ostr_ << ')';
}
```

- We could insert a `print` method
- But that's not nice
- We can use the `operator<<`
- But we no longer can pass additional arguments
- Unless...

Sugaring the PrettyPrinter

```
void
PrettyPrinter::operator()(const Bin& e)
  override
{
  ostr_ << '(';
  e.lhs()->accept(*this);
  ostr_ << e.oper();
  e.rhs()->accept(*this);
  ostr_ << ')';
}
```

- We could insert a `print` method
- But that's not nice
- We can use the `operator<<`
- But we no longer can pass additional arguments
- Unless...

Sugaring the PrettyPrinter

```
void
PrettyPrinter::operator()(const Bin& e)
  override
{
  ostr_ << '(';
  e.lhs()->accept(*this);
  ostr_ << e.oper();
  e.rhs()->accept(*this);
  ostr_ << ')';
}
```

- We could insert a `print` method
- But that's not nice
- We can use the `operator<<`
- But we no longer can pass additional arguments
- Unless...

Sugaring the PrettyPrinter

```
void
PrettyPrinter::operator()(const Bin& e)
  override
{
  ostr_ << '(';
  e.lhs()->accept(*this);
  ostr_ << e.oper();
  e.rhs()->accept(*this);
  ostr_ << ')';
}
```

- We could insert a `print` method
- But that's not nice
- We can use the `operator<<`
- But we no longer can pass additional arguments
- Unless... we can put data in the stream

Visitors Hierarchies

- Implement default behaviors (`DefaultVisitor`, `DefaultConstVisitor`)
- Overloaded virtual member functions must be imported.

```
class Renamer : public DefaultVisitor
{
public:
  using super_t = DefaultVisitor;
  using super_t::operator();
  //...
}
```

Visitors Hierarchies

- Implement default behaviors (`DefaultVisitor`, `DefaultConstVisitor`)
- Overloaded virtual member functions must be imported.

```
class Renamer : public DefaultVisitor
{
public:
  using super_t = DefaultVisitor;
  using super_t::operator();
  //...
}
```

Visitors Hierarchies

- Specialize behaviors

  `DesugarVisitor <: Cloner`,
  `overload::TypeChecker <: type::TypeChecker`, ...

```
void TypeChecker::operator()(ast::LetExp& e) override
{
  // The type of a LetExp is that of its body.
  super_t::operator()(e);
  type_default(e, type(e.body_get()));
}
```

- Use C++ templates to factor
  (e.g., `DefaultVisitor` and `DefaultConstVisitor`)

Visitors Hierarchies

- Specialize behaviors

  `DesugarVisitor <: Cloner`,

  `overload::TypeChecker <: type::TypeChecker`, ...

```
void TypeChecker::operator()(ast::LetExp& e) override
{
  // The type of a LetExp is that of its body.
  super_t::operator()(e);
  type_default(e, type(e.body_get()));
}
```

- Use C++ templates to factor

  (e.g., `DefaultVisitor` and `DefaultConstVisitor`)

Visitor Combinators

- Work and traversal are still too heavily interrelated

→ Create visitors from basic traversal bricks: *combinators* [Visser, 2001].

Combinator	Description
Identity	Do nothing.
Sequence(v_1, v_2)	Sequentially run visitor v_1 then v_2.
Fail	Raise an exception.
Choice(v_1, v_2)	Try visitor v_1; if v_1 fails, try v_2.
All(v)	Apply visitor v sequentially to every immediate subtree.
One(v)	Apply visitor v sequentially to the immediate subtrees until it succeeds.

Visitor Combinators (cont.)

- Combine them to create visiting strategies.

$$
\begin{aligned}
\text{Twice}(v) &:= \text{Sequence}(v, v) \\
\text{Try}(v) &:= \text{Choice}(v, \mathit{Identity}) \\
\text{TopDown}(v) &:= \text{Sequence}(v, \text{All}(\text{TopDown}(v))) \\
\text{BottomUp}(v) &:= \text{Sequence}(\text{All}(\text{BottomUp}(v)), v)
\end{aligned}
$$

Applications

Desugaring

Syntactic Sugar in Lambda-Calculus

Curryfication $\lambda xy.e \Rightarrow \lambda x.(\lambda y.e)$

Local variables $\text{let } x = e_1 \text{ in } e_2 \Rightarrow (\lambda x.e_2).e_1$

Core Languages A sound basis.

Syntactic Sugar in Lambda-Calculus

Curryfication $\lambda xy.e \Rightarrow \lambda x.(\lambda y.e)$

Local variables $\text{let } x = e_1 \text{ in } e_2 \Rightarrow (\lambda x.e_2).e_1$

Core Languages A sound basis.

Syntactic Sugar in Lambda-Calculus

Curryfication	$\lambda xy.e \Rightarrow \lambda x.(\lambda y.e)$
Local variables	$\text{let } x = e_1 \text{ in } e_2 \Rightarrow (\lambda x.e_2).e_1$
Core Languages	A sound basis.

List Comprehension in Haskell

Quicksort in Haskell

```
qsort []     = []
qsort (x:xs) = qsort lt_x ++ [x] ++ qsort ge_x
               where lt_x = [y | y <- xs, y <  x]
                     ge_x = [y | y <- xs, x <= y]
```

List Comprehension in Haskell

Sugared

```
[(x,y) | x <- [1 .. 6], y <- [1 .. x], x+y < 10]
```

Desugared

```
filter p (concat (map (\ x -> map (\ y -> (x,y))
                      [1..x]) [1..6]))
  where p (x,y) = x+y < 10
```

List Comprehension in Haskell

Sugared

```
[(x,y) | x <- [1 .. 6], y <- [1 .. x], x+y < 10]
```

Desugared

```
filter p (concat (map (\ x -> map (\ y -> (x,y))
                          [1..x]) [1..6]))
    where p (x,y) = x+y < 10
```

Desugaring

- Interferences with error messages, e.g., during type checking:

```
% echo '"true" | 42' | tc -T -
standard input:1.1-6: type mismatch
  condition type: string
  expected type: int
```

- The code the type-checker actually saw:

```
% echo '"true" | 42' | tc -XA -
/* == Abstract Syntax Tree. == */

function _main() =
  (
    (if "true"
      then 1
      else (42 <> 0));
    ()
  )
```

- Similarly with CPP

Desugaring

- Interferences with error messages, e.g., during type checking:

```
% echo '"true" | 42' | tc -T -
standard input:1.1-6: type mismatch
  condition type: string
  expected type: int
```

- The code the type-checker actually saw:

```
% echo '"true" | 42' | tc -XA -
/* == Abstract Syntax Tree. == */

function _main() =
  (
    (if "true"
      then 1
      else (42 <> 0));
    ()
  )
```

- Similarly with CPP

Desugaring

- Interferences with error messages, e.g., during type checking:

```
% echo '"true" | 42' | tc -T -
standard input:1.1-6: type mismatch
  condition type: string
  expected type: int
```

- The code the type-checker actually saw:

```
% echo '"true" | 42' | tc -XA -
/* == Abstract Syntax Tree. == */

function _main() =
  (
    (if "true"
      then 1
      else (42 <> 0));
    ()
  )
```

- Similarly with CPP

Existing Tools

ast Generators

- built in generation of various hooks, including for visitors
- generation of visitor skeletons

Treecc

[Weatherley, 2002]

The approach that we take with "treecc" is similar to that used by "yacc". A simple rule-based language is devised that is used to describe the intended behaviour declaratively. Embedded code is used to provide the specific implementation details. A translator then converts the input into source code that can be compiled in the usual fashion.

The translator is responsible for generating the tree building and walking code, and for checking that all relevant operations have been implemented on the node types. Functions are provided that make it easier to build and walk the tree data structures from within a "yacc" grammar and other parts of the compiler.

Treecc

[Weatherley, 2002]

The approach that we take with "treecc" is similar to that used by "yacc". A simple rule-based language is devised that is used to describe the intended behaviour declaratively. Embedded code is used to provide the specific implementation details. A translator then converts the input into source code that can be compiled in the usual fashion.

The translator is responsible for generating the tree building and walking code, and for checking that all relevant operations have been implemented on the node types. Functions are provided that make it easier to build and walk the tree data structures from within a "yacc" grammar and other parts of the compiler.

Treecc: a simple example for expressions

Yacc grammar

Example from [The DotGNU Project, 2009].

```
%token INT FLOAT

%%

expr: INT
    | FLOAT
    | '(' expr ')'
    | expr '+' expr
    | expr '-' expr
    | expr '*' expr
    | expr '/' expr
    | '-' expr
    ;
```

Treecc: a simple example for expressions (cont).

```
%node expression %abstract %typedef
%node binary expression %abstract = {
  expression* expr1;
  expression* expr2;
}
%node unary expression %abstract = {
  expression* expr;
}
%node intnum expression = {
  int num;
}
%node floatnum expression = {
  float num;
}
%node plus binary
%node minus binary
%node multiply binary
%node divide binary
%node negate unary
```

Treecc: a simple example for expressions

Yacc grammar augmented to build the parse tree

```
%union {
    expression* node;    int inum;    float fnum;
}
%token <inum> INT
%token <fnum> FLOAT
%type <node> expr
%%
expr: INT                   { $$ = intnum_create($1); }
    | FLOAT                 { $$ = floatnum_create($1); }
    | '(' expr ')'          { $$ = $2; }
    | expr '+' expr         { $$ = plus_create($1, $3); }
    | expr '-' expr         { $$ = minus_create($1, $3); }
    | expr '*' expr         { $$ = multiply_create($1, $3); }
    | expr '/' expr         { $$ = divide_create($1, $3); }
    | '-' expr              { $$ = negate_create($2); }
    ;
```

The Introspector

Extract meta-data about programs (from compiler, build & make system, savannah/sourceforge management, packaging system, version control tools and mailing lists) and present it to you for making your job as a programmer easier.

The software is free software in the spirit of the GNU manifesto and is revolutionary in the freedoms that it intends on granting to its users.

Originally the GCC "C" compiler, but supports Perl, Bison, M4, Bash, C#, Java, C++, Fortran, Objective C, Lisp and Scheme. [DuPont, 2004]

According to some, a threat to Free Software.

The Introspector

Extract meta-data about programs (from compiler, build & make system, savannah/sourceforge management, packaging system, version control tools and mailing lists) and present it to you for making your job as a programmer easier.

The software is free software in the spirit of the GNU manifesto and is revolutionary in the freedoms that it intends on granting to its users.

Originally the GCC "C" compiler, but supports Perl, Bison, M4, Bash, C#, Java, C++, Fortran, Objective C, Lisp and Scheme. [DuPont, 2004]

According to some, a threat to Free Software.

The Introspector

Extract meta-data about programs (from compiler, build & make system, savannah/sourceforge management, packaging system, version control tools and mailing lists) and present it to you for making your job as a programmer easier.

The software is free software in the spirit of the GNU manifesto and is revolutionary in the freedoms that it intends on granting to its users.

Originally the GCC "C" compiler, but supports Perl, Bison, M4, Bash, C#, Java, C++, Fortran, Objective C, Lisp and Scheme. [DuPont, 2004]

According to some, a threat to Free Software.

The Introspector

Extract meta-data about programs (from compiler, build & make system, savannah/sourceforge management, packaging system, version control tools and mailing lists) and present it to you for making your job as a programmer easier.

The software is free software in the spirit of the GNU manifesto and is revolutionary in the freedoms that it intends on granting to its users.

Originally the GCC "C" compiler, but supports Perl, Bison, M4, Bash, C#, Java, C++, Fortran, Objective C, Lisp and Scheme. [DuPont, 2004]

According to some, a threat to Free Software.

GCC-XML

C++ has become a popular and powerful language, but parsing it is a very challenging problem. This has discouraged the development of tools meant to work directly with the language.

There is one open-source C++ parser, the C++ front-end to GCC, which is currently able to deal with the language in its entirety. The purpose of the GCC-XML extension is to generate an XML description of a C++ program from GCC's internal representation.

Since XML is easy to parse, other development tools will be able to work with C++ programs without the burden of a complicated C++ parser. [King, 2004]

GCC-XML is no longer maintained but replaced by CASTXML

GCC-XML

C++ has become a popular and powerful language, but parsing it is a very challenging problem. This has discouraged the development of tools meant to work directly with the language.

There is one open-source C++ parser, the C++ front-end to GCC, which is currently able to deal with the language in its entirety. The purpose of the GCC-XML extension is to generate an XML description of a C++ program from GCC's internal representation.

Since XML is easy to parse, other development tools will be able to work with C++ programs without the burden of a complicated C++ parser. [King, 2004]

GCC-XML is no longer maintained but replaced by CASTXML

GCC-XML

C++ has become a popular and powerful language, but parsing it is a very challenging problem. This has discouraged the development of tools meant to work directly with the language.

There is one open-source C++ parser, the C++ front-end to GCC, which is currently able to deal with the language in its entirety. The purpose of the GCC-XML extension is to generate an XML description of a C++ program from GCC's internal representation.

Since XML is easy to parse, other development tools will be able to work with C++ programs without the burden of a complicated C++ parser. [King, 2004]

GCC-XML is no longer maintained but replaced by CASTXML

GCC-XML

C++ has become a popular and powerful language, but parsing it is a very challenging problem. This has discouraged the development of tools meant to work directly with the language.

There is one open-source C++ parser, the C++ front-end to GCC, which is currently able to deal with the language in its entirety. The purpose of the GCC-XML extension is to generate an XML description of a C++ program from GCC's internal representation.

Since XML is easy to parse, other development tools will be able to work with C++ programs without the burden of a complicated C++ parser. [King, 2004]

GCC-XML is no longer maintained but replaced by CASTXML

The Case of the Tiger Compiler

The ast

Tiger Abstract Syntax

```
/Ast/                  (Location location)
  /Exp/                ()
*    ArrayExp
*    AssignExp
*    BreakExp
*    CallExp
*      MethodCallExp
     CastExp           (Exp exp, Ty ty)
     ForExp            (VarDec vardec, Exp hi, Exp body)
*    IfExp
     IntExp            (int value)
*    LetExp
     NilExp            ()
*    ObjectExp
     OpExp             (Exp left, Oper oper, Exp right)
*    RecordExp
*    SeqExp
*    StringExp
     WhileExp          (Exp test, Exp body)
```

Tiger Abstract Syntax

```
/Ast/                 (Location location)
  /Exp/               ()
*   /Var/
      CastVar         (Var var, Ty ty)
*     FieldVar
      SimpleVar       (symbol name)
      SubscriptVar    (Var var, Exp index)

  /Dec/               (symbol name)
    FunctionDec       (VarDecs formals, NameTy result, Exp body)
      MethodDec       ()
    TypeDec           (Ty ty)
    VarDec            (NameTy type_name, Exp init)

  /Ty/                ()
    ArrayTy           (NameTy base_type)
    ClassTy           (NameTy super, DecsList decs)
    NameTy            (symbol name)
*   RecordTy
```

Tiger Abstract Syntax

```
DecsList        (decs_type decs)

Field           (symbol name, NameTy type_name)

FieldInit       (symbol name, Exp init)
```

Tiger Abstract Syntax

Some of these classes also derive from other classes.

```
/Escapable/
  VarDec              (NameTy type_name, Exp init)

/Typable/
  /Dec/               (symbol name)
  /Exp/               ()
  /Ty/                ()

/TypeConstructor/
  /Ty/                ()
  FunctionDec         (VarDecs formals, NameTy result, Exp body)
  TypeDec             (Ty ty)
```

Syntactic Sugar

Tiger Sugar

Light
- `if then`

Regular
- Unary -
- `&` and `|`
- Beware of `( exp )` vs. `( exps )`
- Declarations (Types and Functions)

Extra
- `for`
- `?:` as in GNU C (`a ?: b`)
- `where`
- Function overload

Tiger Sugar

Light
- `if then`

Regular
- Unary -
- & and |
- Beware of `( exp )` vs. `( exps )`
- Declarations (Types and Functions)

Extra
- `for`
- `?:` as in GNU C (`a ?: b`)
- `where`
- Function overload

Tiger Sugar

Light
- `if then`

Regular
- Unary -
- `&` and `|`
- Beware of `( exp )` vs. `( exps )`
- Declarations (Types and Functions)

Extra
- `for`
- `?:` as in GNU C (`a ?: b`)
- `where`
- Function overload

Desugaring

Desugaring in Abstract Syntax

```
exp: exp "&" exp
{
  $$ = new IfExp(@$, $1,
                 new OpExp(@$,$3, OpExp::ne, new IntExp(@2, 0)),
                 new IntExp(@2, 0));
}
```

Desugaring in Concrete Syntax

```
exp: exp "&" exp
{
  $$ = parse::parse(parse::Tweast() <<
    "if " << $1 << " then " << $3 << "<> 0 else 0");
}
```

Tweast: Text With Embedded Abstract Syntax Trees

Desugaring

Desugaring in Abstract Syntax

```
exp: exp "&" exp
{
  $$ = new IfExp(@$, $1,
                 new OpExp(@$,$3, OpExp::ne, new IntExp(@2, 0)),
                 new IntExp(@2, 0));
}
```

Desugaring in Concrete Syntax

```
exp: exp "&" exp
{
  $$ = parse::parse(parse::Tweast() <<
     "if " << $1 << " then " << $3 << "<> 0 else 0");
}
```

Tweast: Text With Embedded Abstract Syntax Trees

Desugaring

Desugaring in Abstract Syntax

```
exp: exp "&" exp
{
  $$ = new IfExp(@$, $1,
                 new OpExp(@$,$3, OpExp::ne, new IntExp(@2, 0)),
                 new IntExp(@2, 0));
}
```

Desugaring in Concrete Syntax

```
exp: exp "&" exp
{
  $$ = parse::parse(parse::Tweast() <<
     "if " << $1 << " then " << $3 << "<> 0 else 0");
}
```

Tweast: Text With Embedded Abstract Syntax Trees

Visitors

Stubs in ast nodes

Every single AST node needs accept.

ast/let-exp.cc

```
void LetExp::accept(ConstVisitor& v) const
{
  v(*this);
}

void LetExp::accept(Visitor& v)
{
  v(*this);
}
```

This can be factored by inheritance [Alexandrescu, 2001].

Inheritance to Factor (Mixin)

parse/metavar-map.hh

```
template <typename Data>
struct MetavarMap
{
  /// Append a metavariable.
  void append_(int k,
               Data* d);
  /// Extract a metavariable.
  Data* take_(int k);
  /// Metavariables.
  map<int, Data*> map_;
};
```

parse/tweast.cc

```
class Tweast
  : public MetavarMap<Exp>
  , public MetavarMap<Var>
  , public MetavarMap<NameTy>
  , public MetavarMap<DecsList>
{
  // ...
};
```

Inheritance to Factor (Mixin)

parse/metavar-map.hh

```
template <typename Data>
struct MetavarMap
{
  /// Append a metavariable.
  void append_(int k,
               Data* d);
  /// Extract a metavariable.
  Data* take_(int k);
  /// Metavariables.
  map<int, Data*> map_;
};
```

parse/tweast.cc

```
class Tweast
  : public MetavarMap<Exp>
  , public MetavarMap<Var>
  , public MetavarMap<NameTy>
  , public MetavarMap<DecsList>
{
  // ...
};
```

Visitors

PrettyPrinter	Pretty-printer
Binder	Bind uses to declarations
Renamer	Unique names
TypeChecker	Annotate nodes with their type
object::Binder	Bind for Object Tiger
object::TypeChecker	Check types for Object Tiger
overload::Binder	Bind for overloaded Tiger
overload::TypeChecker	Check types for overloaded Tiger

Visitors

PrettyPrinter	Pretty-printer
Binder	Bind uses to declarations
Renamer	Unique names
TypeChecker	Annotate nodes with their type
object::Binder	Bind for Object Tiger
object::TypeChecker	Check types for Object Tiger
overload::Binder	Bind for overloaded Tiger
overload::TypeChecker	Check types for overloaded Tiger

Visitors

PrettyPrinter	Pretty-printer
Binder	Bind uses to declarations
Renamer	Unique names
TypeChecker	Annotate nodes with their type
object::Binder	Bind for Object Tiger
object::TypeChecker	Check types for Object Tiger
overload::Binder	Bind for overloaded Tiger
overload::TypeChecker	Check types for overloaded Tiger

Visitors

PrettyPrinter	Pretty-printer
Binder	Bind uses to declarations
Renamer	Unique names
TypeChecker	Annotate nodes with their type
object::Binder	Bind for Object Tiger
object::TypeChecker	Check types for Object Tiger
overload::Binder	Bind for overloaded Tiger
overload::TypeChecker	Check types for overloaded Tiger

Visitors

PrettyPrinter	Pretty-printer
Binder	Bind uses to declarations
Renamer	Unique names
TypeChecker	Annotate nodes with their type
object::Binder	Bind for Object Tiger
object::TypeChecker	Check types for Object Tiger
overload::Binder	Bind for overloaded Tiger
overload::TypeChecker	Check types for overloaded Tiger

Visitors

PrettyPrinter	Pretty-printer
Binder	Bind uses to declarations
Renamer	Unique names
TypeChecker	Annotate nodes with their type
object::Binder	Bind for Object Tiger
object::TypeChecker	Check types for Object Tiger
overload::Binder	Bind for overloaded Tiger
overload::TypeChecker	Check types for overloaded Tiger

Visitors

PrettyPrinter	Pretty-printer
Binder	Bind uses to declarations
Renamer	Unique names
TypeChecker	Annotate nodes with their type
object::Binder	Bind for Object Tiger
object::TypeChecker	Check types for Object Tiger
overload::Binder	Bind for overloaded Tiger
overload::TypeChecker	Check types for overloaded Tiger

Visitors

PrettyPrinter	Pretty-printer
Binder	Bind uses to declarations
Renamer	Unique names
TypeChecker	Annotate nodes with their type
object::Binder	Bind for Object Tiger
object::TypeChecker	Check types for Object Tiger
overload::Binder	Bind for overloaded Tiger
overload::TypeChecker	Check types for overloaded Tiger

Visitors

object::DesugarVisitor	Desugar Object Tiger code into the non-object core
DesugarVisitor	Handling syntactic sugar
BoundCheckingVisitor	Bounds checking
Inliner	Function inlining
Pruner	Remove useless function definitions
EscapesVisitor	Escaping variables
Translator	Conversion to HIR

Visitors

object::DesugarVisitor	Desugar Object Tiger code into the non-object core
DesugarVisitor	Handling syntactic sugar
BoundCheckingVisitor	Bounds checking
Inliner	Function inlining
Pruner	Remove useless function definitions
EscapesVisitor	Escaping variables
Translator	Conversion to HIR

Visitors

object::DesugarVisitor	Desugar Object Tiger code into the non-object core
DesugarVisitor	Handling syntactic sugar
BoundCheckingVisitor	Bounds checking
Inliner	Function inlining
Pruner	Remove useless function definitions
EscapesVisitor	Escaping variables
Translator	Conversion to HIR

Visitors

object::DesugarVisitor	Desugar Object Tiger code into the non-object core
DesugarVisitor	Handling syntactic sugar
BoundCheckingVisitor	Bounds checking
Inliner	Function inlining
Pruner	Remove useless function definitions
EscapesVisitor	Escaping variables
Translator	Conversion to HIR

Visitors

object::DesugarVisitor	Desugar Object Tiger code into the non-object core
DesugarVisitor	Handling syntactic sugar
BoundCheckingVisitor	Bounds checking
Inliner	Function inlining
Pruner	Remove useless function definitions
EscapesVisitor	Escaping variables
Translator	Conversion to HIR

Visitors

object::DesugarVisitor	Desugar Object Tiger code into the non-object core
DesugarVisitor	Handling syntactic sugar
BoundCheckingVisitor	Bounds checking
Inliner	Function inlining
Pruner	Remove useless function definitions
EscapesVisitor	Escaping variables
Translator	Conversion to HIR

Visitors

object::DesugarVisitor	Desugar Object Tiger code into the non-object core
DesugarVisitor	Handling syntactic sugar
BoundCheckingVisitor	Bounds checking
Inliner	Function inlining
Pruner	Remove useless function definitions
EscapesVisitor	Escaping variables
Translator	Conversion to HIR

Bibliography I


Alexandrescu, A. (2001).
Modern C++ Design: Generic Programming and Design Patterns Applied.
Addison-Wesley.


ASN.1 Consortium (2003).
The ASN.1 Consortium.
http://www.asn1.org/.


Centrum voor Wiskunde en Informatica (2004).
The ATerm Library.
http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/ATermLibrary.


Dubuisson, O. (2003).
The ASN.1 Information Site.
http://asn1.elibel.tm.fr/en/.

Bibliography II


DuPont, J. M. (2004).
The Introspector Project.
http://introspector.sourceforge.net/.


King, B. (2004).
Gcc-xml.
http://gccxml.org.


The DotGNU Project (2009).
Tree Compiler-Compiler.
http://dotgnu.org/treecc/treecc.html.

Bibliography III

Visser, E. (1995).
A family of syntax definition formalisms.
In van den Brand, M. G. J. et al., editors, *ASF+SDF'95. A Workshop on Generating Tools from Algebraic Specifications*, pages 89–126. Technical Report P9504, Programming Research Group, University of Amsterdam.

Visser, J. (2001).
Visitor combination and traversal control.
ACM SIGPLAN Notices, 36(11):270–282.
OOPSLA 2001 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications.

Weatherley, R. (2002).
Treecc: An aspect-oriented approach to writing compilers.
`http://dotgnu.org/treecc_essay.html`.

Bibliography IV


Yu, Y. and D'Hollander, E. (2003).
YAXX: YAcc eXtension to XML, a user manual.
http://yaxx.sourceforge.net/.