Compiler Construction

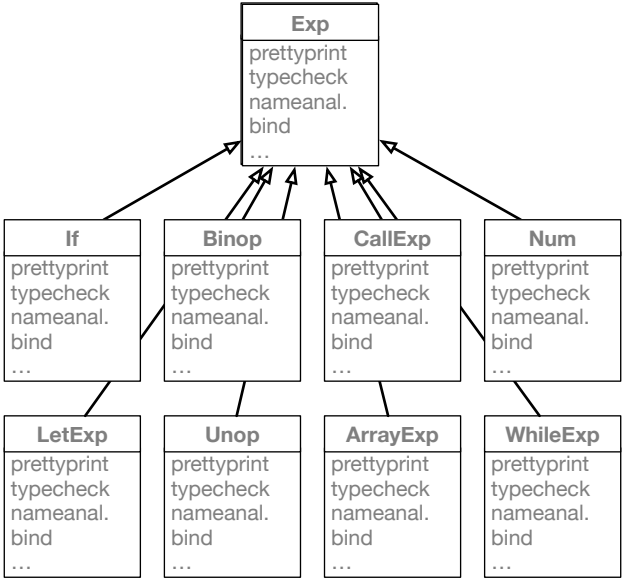
Efficient Traversals

Traversal in compilers

- Pretty-printer
- Name analysis
- Unique identifiers
- Desugaring
- Type Checking

- Escaping variables
- Inlining
- High level optimization
- Translation to intermediate representation
- ...

Inheritance Problem (1/2)



Inheritance Problem (2/2)

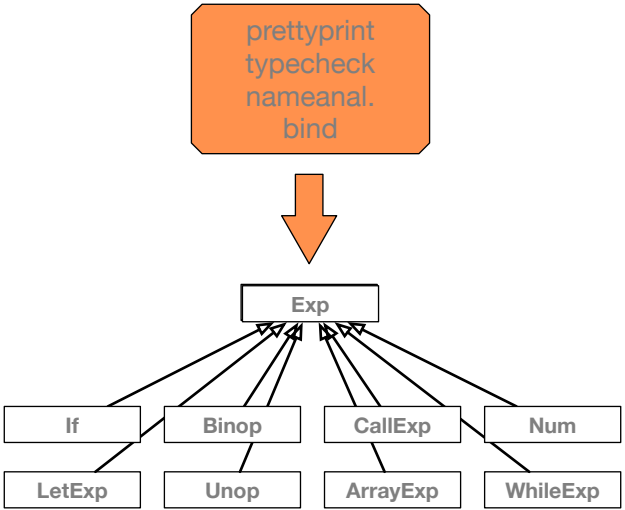
Inheritance is not adapted

- Obfuscate AST classes
  ⇒ hard to maintain
  ⇒ can no longer be generated
- Spread traversal code into multiple files ⇒ Error-prone

Is there a solution?

Yes, but non trivial
⇒ Build external processing

External processing



First attempt: pretty-printer (1/3)

Use external processing using C++ print operator (<<)

Modifications for the AST:

1. Only declare operator in classes

```
friend ostream& operator<<(ostream& o, const Class& c);
```

2. Then write the external processing!

First attempt: pretty-printer(2/3)

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

ostream& operator<<(ostream& o, const Bin& tree) {
  return o
    << '(' << *tree.lhs_
    << *tree.oper_
    << *tree.rhs_ << ')'
  ;
}

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

First attempt: pretty-printer (3/3)

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

output

```
Exp: Uh oh...
Bin: (Uh oh...+Uh oh...)
```

Problem description

- **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++

Problem description

- **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++

How to process the AST externally while exploiting dynamic dispatch?

Discussion (1/2)

Ask the hierarchy to dispatch

Discussion (2/2)

We could reuse virtual method strategy:

- Augment all classes with a **virtual print**
- Call **print** inside of **operator«**

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

Works but not external anymore (still needs a `print` method in every class) and still obfuscates the AST

Solution (1/2)

Separate processing and dispatching

Solution (1/2)

Separate processing and dispatching

Processing

- Keep it external
- Add new processings easily

Dispatching

- Keep it internal
- All processings will be dispatched in the same way: **factorize it!**

Solution (2/2)

Multimethods

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

```
using std::ostream;

ostream& operator<<(ostream& o, virtual Exp& e);
ostream& operator<<(ostream& o, virtual Bin& e);
ostream& operator<<(ostream& o, virtual Num& e);
```

Available in CLOS, but not in C++

Multimethods in C++

No multimethods in C++11, 14, 17, 20,...

Simulate via a **trampoline**

- Instrument every AST class once.
- With this instrumentation we are able to detect the dynamic type
- .. and to process externally!

Naive Implementation (1/4)

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

  // Define two new types for callbacks
  using bin_cb = std::function<auto (const Bin&) -> void>;
  using num_cb = std::function<auto (const Num&) -> void>;

  // Write a "dispatcher" using virtual
  virtual void dispatch(bin_cb bin, num_cb num) const = 0;
};
```

Naive Implementation (2/4)

```
void Bin::dispatch(bin_cb bin, num_cb) const
{
  bin(*this);
}

void Num::dispatch(bin_cb, num_cb num) const
{
  num(*this);
}
```

Naive Implementation (3/4)

```
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;
}
```

Naive Implementation (4/4)

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

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

Summary