From parsing to sema: making sense of syntax trees

Michał Dominiak
Nokia Networks
griwes@griwes.info

Outline

Parser

- converts a program from source code into a structured version of it

Parser

- converts a program from source code into a structured version of it
- produces an AST – abstract syntax tree

Parser

- converts a program from source code into a structured version of it
- produces an AST – abstract syntax tree
- detects invalid uses of language syntax

Parser

- converts a program from source code into a structured version of it
- produces an AST – abstract syntax tree
- detects invalid uses of language syntax
- if you want to hear more – watch my previous talk, "Simple hand written parsers", or reach into books

Semantic analyzer

- converts an AST into an annotated form

Semantic analyzer

- converts an AST into an annotated form
- resolves names

Semantic analyzer

- converts an AST into an annotated form
- resolves names
- type checks the program

Semantic analyzer

- converts an AST into an annotated form
- resolves names
- type checks the program
- possibly validates further aspects of the language

Semantic analyzer

- converts an AST into an annotated form
- resolves names
- type checks the program
- possibly validates further aspects of the language
- this is the subject of this talk

The curious case of C++

The curious case of C++

```
a * b;
```

The curious case of C++

```
struct a;

a * b;
```

The curious case of C++

```
int a, b;

a * b;
```

The curious case of C++

- C++ (same as C) has an at least context-sensitive grammar (type 1 in Chomsky hierarchy)

The curious case of C++

- C++ (same as C) has an at least context-sensitive grammar (type 1 in Chomsky hierarchy)
- some argue it is even unrestricted (type 0)

The curious case of C++

- C++ (same as C) has an at least context-sensitive grammar (type 1 in Chomsky hierarchy)
- some argue it is even unrestricted (type 0)
- parsing C++ without doing *some* semantic analysis at the same time is not something you want to do

The curious case of C++

- C++ (same as C) has an at least context-sensitive grammar (type 1 in Chomsky hierarchy)
- some argue it is even unrestricted (type 0)
- parsing C++ without doing *some* semantic analysis at the same time is not something you want to do
- `typename` and `template` disambiguators – `typename T::type`, `T::template foo<1>`, `tuple.template get<3>()`

Optimizer

- simplifies the program

Optimizer

- simplifies the program
- sounds simple, doesn't it?

Optimizer

- simplifies the program
- sounds simple, doesn't it?
- I don't dare entertain the idea of giving a talk on this ;)

Code generator

- generates the code for consumption by the next program in the pipeline

Code generator

- generates the code for consumption by the next program in the pipeline
- usually some flavor of assembly, or IR (thanks, LLVM!)

Code generator

- generates the code for consumption by the next program in the pipeline
- usually some flavor of assembly, or IR (thanks, LLVM!)
- seems compiler-centric, but may also be included in an interpreter (think JITs)

Code generator

- generates the code for consumption by the next program in the pipeline
- usually some flavor of assembly, or IR (thanks, LLVM!)
- seems compiler-centric, but may also be included in an interpreter (think JITs)
- probably (?) not as interesting as a talk than the previous ones?

Execution engine

- generally found in interpreters...

Execution engine

- generally found in interpreters...
- ...but ask your local C++ compiler developer about `constexpr`

Execution engine

- generally found in interpreters...
- ...but ask your local C++ compiler developer about `constexpr`
- this talk touches on this topic, but not significantly

Outline

1. Structure of compilers and interpreters
2. Creating ASTs
3. Processing ASTs
4. What I tried in an interpreter
5. What I tried in a compiler
6. Real world

Dynamic polymorphism

- easy to extend dynamically

Dynamic polymorphism

- easy to extend dynamically
- use virtual dispatch for visitation

Dynamic polymorphism

- easy to extend dynamically
- use virtual dispatch for visitation
- or use dynamic casts for visitation

Dynamic polymorphism

```
struct visitor {
    virtual bool visit(const assignment &) {}
};

struct expression {
    virtual ~expression() = default;
    virtual bool visit(visitor & v) = 0;
};

struct assignment : expression {
    virtual bool visit(visitor & v) override {
        return visitor.visit(*this);
    }
};
```

Dynamic polymorphism

```
struct expression {
    virtual ~expression() = default;
};
struct assignment : expression {};

struct visitor {
    std::unordered_map<std::type_index, std::function<void (expression *)>> callbacks;
    bool visit(expression * expr) {
        return callbacks.at(typeid(expr))(expr);
    }
    template<typename T, typename F>
    void add_visitor(F && f) {
        callbacks.emplace(typeid(T), [f = std::forward<F>(f)](expression * ptr) {
            f(dynamic_cast<T *>(ptr));
        });
    }
};
```

Static polymorphism

- harder to extend, with lots of recompilation

Static polymorphism

- harder to extend, with lots of recompilation
- type-checked for handling most (all?) cases

Static polymorphism

- harder to extend, with lots of recompilation
- type-checked for handling most (all?) cases
- variant-based

Static polymorphism

```
using expression = variant<
    assignment,
    binary_expression,
    function_call
>;

void analyze_expression(const expression & expr) {
    fmap(expr, make_overload_set(
        [](const assignment &) { /* handle assignment */ },
        [](const binary_expression &) { /* handle binary expression */ },
        [](const function_call &) { /* handle function call */ }
    ));
}
```

So what?

- the way the AST is structured will affect how you consume it in analyzer

So what?

- the way the AST is structured will affect how you consume it in analyzer
- making the AST polymorphic would require either:

So what?

- the way the AST is structured will affect how you consume it in analyzer
- making the AST polymorphic would require either:
  - making the parser AST aware of the analyzer types, or

So what?

- the way the AST is structured will affect how you consume it in analyzer
- making the AST polymorphic would require either:
    - making the parser AST aware of the analyzer types, or
    - using the crazy dynamic visitation scheme to spit out analyzesr tree

Outline

1. Structure of compilers and interpreters
2. Creating ASTs
3. Processing ASTs
4. What I tried in an interpreter
5. What I tried in a compiler
6. Real world

Single pass

- go in order, once

Single pass

- go in order, once
- set itself on fire if anything is not resolved

Multiple passes

- go in order, multiple times

Multiple passes

- go in order, multiple times
- do a set number of multiple passes

Multiple passes

- go in order, multiple times
- do a set number of multiple passes, or
- keep going until everything is resolved

Multiple passes

- go in order, multiple times
- do a set number of multiple passes, or
- keep going until everything is resolved ...or report an error once a pass makes no progress

DAG of AST elements

- build a tree of (possibly incomplete) elements

DAG of AST elements

- build a tree of (possibly incomplete) elements
- gather information about the dependencies

DAG of AST elements

- build a tree of (possibly incomplete) elements
- gather information about the dependencies
- execute the DAG of dependencies

DAG of AST elements

- build a tree of (possibly incomplete) elements
- gather information about the dependencies
- execute the DAG of dependencies
- (shared) futures are great for this!

DAG of AST elements

- build a tree of (possibly incomplete) elements
- gather information about the dependencies
- execute the DAG of dependencies
- (shared) futures are great for this!
- coroutines will make them *better* at it

Annotating in place

- avoids constant reallocation

Annotating in place

- avoids constant reallocation
- possibly easier to debug, since addresses don't change

Annotating in place

- avoids constant reallocation
- possibly easier to debug, since addresses don't change
- to an extend – a necessity for C++, since you are analyzing during parsing

Generating a new structure

- allows for a higher degree of encapsulation between the parser and the analyzer

Generating a new structure

- allows for a higher degree of encapsulation between the parser and the analyzer
- allows for more flexibility with the analyzed structure

Outline

Language

```
foo = bar
baz = "123"
fizz = buzz(bar, "123")
bar = "this is " + "text"
```

Initial pass

- string literals: create an element of the analyzed AST

Initial pass

- string literals: create an element of the analyzed AST
- names: try to resolve

Initial pass

- string literals: create an element of the analyzed AST
- names: try to resolve
    - if the name is known - mark resolved and refer to the actual value of the name

Initial pass

- string literals: create an element of the analyzed AST
- names: try to resolve
  - if the name is known - mark resolved and refer to the actual value of the name
  - if the name is not know - save the name for later and return a "delayed expression"

Initial pass

- string literals: create an element of the analyzed AST
- names: try to resolve
  - if the name is known - mark resolved and refer to the actual value of the name
  - if the name is not know - save the name for later and return a "delayed expression"
- addition: analyze operands

Initial pass

- string literals: create an element of the analyzed AST
- names: try to resolve
  - if the name is known - mark resolved and refer to the actual value of the name
  - if the name is not know - save the name for later and return a "delayed expression"
- addition: analyze operands
  - if both operands are resolved - evaluate the operation

Initial pass

- string literals: create an element of the analyzed AST
- names: try to resolve
  - if the name is known - mark resolved and refer to the actual value of the name
  - if the name is not know - save the name for later and return a "delayed expression"
- addition: analyze operands
  - if both operands are resolved - evaluate the operation
  - if any one of the operands are not resolved - save the arguments for later and return a (different kind of a) "delayed expression"

Initial pass

- string literals: create an element of the analyzed AST
- names: try to resolve
  - if the name is known - mark resolved and refer to the actual value of the name
  - if the name is not know - save the name for later and return a "delayed expression"
- addition: analyze operands
  - if both operands are resolved - evaluate the operation
  - if any one of the operands are not resolved - save the arguments for later and return a (different kind of a) "delayed expression"
- type instantiation: analyze the type name, analyze the arguments

Initial pass

- string literals: create an element of the analyzed AST
- names: try to resolve
  - if the name is known - mark resolved and refer to the actual value of the name
  - if the name is not know - save the name for later and return a "delayed expression"
- addition: analyze operands
  - if both operands are resolved - evaluate the operation
  - if any one of the operands are not resolved - save the arguments for later and return a (different kind of a) "delayed expression"
- type instantiation: analyze the type name, analyze the arguments
  - if everything is already resolved - evaluate the instantiation

Initial pass

- string literals: create an element of the analyzed AST
- names: try to resolve
  - if the name is known - mark resolved and refer to the actual value of the name
  - if the name is not know - save the name for later and return a "delayed expression"
- addition: analyze operands
  - if both operands are resolved - evaluate the operation
  - if any one of the operands are not resolved - save the arguments for later and return a (different kind of a) "delayed expression"
- type instantiation: analyze the type name, analyze the arguments
  - if everything is already resolved - evaluate the instantiation
  - if anything is not resolved yet - create a (yet another kind of a) "delayed expression"

Initial pass

```
struct _delayed_instantiation_info {
    type_identifier actual_type;
    std::vector<std::shared_ptr<variable>> arguments;
};
struct _delayed_reference_info {
    std::vector<std::u32string> referenced_id_expression;
};
struct _delayed_type_info {
    std::vector<std::u32string> type_name;
    std::vector<std::shared_ptr<variable>> arguments;
};
struct _delayed_operation_info {
    std::shared_ptr<variable> lhs;
    std::shared_ptr<variable> rhs;
    operation_type operation;
};
std::variant<
    std::shared_ptr<variable>,
    _delayed_instantiation_info,
    _delayed_reference_info,
    _delayed_type_info,
    _delayed_operation_info
> _state;
```

Resolving loop

- save the count of currently unresolved expressions

Resolving loop

- save the count of currently unresolved expressions
- iterate through the unresolved expressions, asking them to try to resolve themselves

Resolving loop

- save the count of currently unresolved expressions
- iterate through the unresolved expressions, asking them to try to resolve themselves
- if the count of currently unresolved expressions is different than the saved one, continue

Resolving loop

- save the count of currently unresolved expressions
- iterate through the unresolved expressions, asking them to try to resolve themselves
- if the count of currently unresolved expressions is different than the saved one, continue
- validate if all expressions are resolved

Resolving loop

```
std::size_t previous = 0;
while (previous != ctx.unresolved.size()) {
    previous = ctx.unresolved.size();
    for (auto && u : ctx.unresolved) {
        u.first->try_resolve(ctx);
    }
}

if (!ctx.unresolved.empty()) {
    throw analysis_error{};
}
```

Outline

1. Structure of compilers and interpreters
2. Creating ASTs
3. Processing ASTs
4. What I tried in an interpreter
5. What I tried in a compiler
6. Real world

Attempt #1: mimic the parser

```
struct expression {
    range_type range;
    variant<
        literal<lexer::token_type::string>,
        literal<lexer::token_type::integer>,
        postfix_expression,
        import_expression,
        recursive_wrapper<lambda_expression>,
        recursive_wrapper<unary_expression>,
        recursive_wrapper<binary_expression>
    > expression_value;
};
```

Attempt #1: mimic the parser

```
class postfix_expression { public: void analyze(); /* ... */ };
class unary_expression { public: void analyze(); /* ... */ };
class binary_expression { public: void analyze(); /* ... */ };
using expression = variant<
    literal,
    variable,
    import_expression,
    postfix_expression,
    unary_expression,
    binary_expression
>;
```

Attempt #1: mimic the parser

```
class postfix_expression { public: void analyze(); /* ... */ };
class unary_expression { public: void analyze(); /* ... */ };
class binary_expression { public: void analyze(); /* ... */ };
using expression = variant<
    literal,
    variable,
    import_expression,
    postfix_expression,
    unary_expression,
    binary_expression
>;
void analyze(expression & expr) {
    fmap(expr, [](auto && v) { v.analyze(); return unit{}; });
}
```

Attempt #2: OO-based polymorphism

- pass #1: create the objects, setup scopes so that name lookup works
- "pass" #2: go top-down analyzing dependencies of all AST nodes, and then analyzing themselves

Attempt #2: OO-based polymorphism

- clear inheritance hierarchy that makes repetition go away

Attempt #2: OO-based polymorphism

- clear inheritance hierarchy that makes repetition go away
- expressions are statements

Attempt #2: OO-based polymorphism

- clear inheritance hierarchy that makes repetition go away
- expressions are statements
- similar kinds of expressions can extend other kinds

Attempt #2: OO-based polymorphism

- clear inheritance hierarchy that makes repetition go away
- expressions are statements
- similar kinds of expressions can extend other kinds
- e.g. `identifier` is an `expression_ref`

Attempt #2: OO-based polymorphism

- three main base classes: `statement`, `type`, and...

Attempt #2: OO-based polymorphism

- three main base classes: `statement`, `type`, and... `variable`

Attempt #2: OO-based polymorphism

- three main base classes: `statement`, `type`, and… `variable`
- expressions had associated variables that represented their value

Attempt #2: OO-based polymorphism

- three main base classes: `statement`, `type`, and... `variable`
- expressions had associated variables that represented their value
- a bunch of classes interconverting between one and the other: `expression_variable`

Attempt #2: OO-based polymorphism

- three main base classes: `statement`, `type`, and... `variable`
- expressions had associated variables that represented their value
- a bunch of classes interconverting between one and the other: `expression_variable`, `variable_expression`

Attempt #2: OO-based polymorphism

- three main base classes: `statement`, `type`, and... `variable`
- expressions had associated variables that represented their value
- a bunch of classes interconverting between one and the other: `expression_variable`, `variable_expression`, `expression_ref_variable`

Attempt #2: OO-based polymorphism

- three main base classes: `statement`, `type`, and... `variable`
- expressions had associated variables that represented their value
- a bunch of classes interconverting between one and the other: `expression_variable`, `variable_expression`, `expression_ref_variable`, `variable_ref_expression`

Attempt #2: OO-based polymorphism

- three main base classes: `statement`, `type`, and... `variable`
- expressions had associated variables that represented their value
- a bunch of classes interconverting between one and the other: `expression_variable`, `variable_expression`, `expression_ref_variable`, `variable_ref_expression`
- got rid of `variable`, never been happier (should've done it much sooner)

Making sense through dependency graphs

```
class expression {
    void set_type(type *);
    type * get_type() const;
    @memoized
    virtual future<void> analyze(analysis_context &);
};
```

Making sense through dependency graphs

```
class expression {
    void set_type(type *);
    type * get_type() const;
    @memoized
    virtual future<void> analyze(analysis_context &);
};

class binary_expression : public expression {
    std::unique_ptr<expression> lhs;
    std::unique_ptr<expression> rhs;
    operator op;

    std::unique_ptr<expression> call_expr;
    future<void> analyze(analysis_context &) override;
};
```

Making sense through dependency graphs

```
future<void> binary_expression::analyze(analysis_context & ctx)
{
    return when_all(
        lhs->analyze(ctx),
        rhs->analyze(ctx)
    )
```

Making sense through dependency graphs

```
future<void> binary_expression::analyze(analysis_context & ctx)
{
    return when_all(
        lhs->analyze(ctx),
        rhs->analyze(ctx)
    )
    .then([&]{
        return resolve_overload(ctx, lhs.get(), rhs.get(), op);
    })
```

Making sense through dependency graphs

```
future<void> binary_expression::analyze(analysis_context & ctx)
{
    return when_all(
        lhs->analyze(ctx),
        rhs->analyze(ctx)
    )
    .then([&]{
        return resolve_overload(ctx, lhs.get(), rhs.get(), op);
    })
    .then([&](std::unique_ptr<expression> call){
        call_expr = std::move(call);
        return call_expr->analyze(ctx);
    })
```

Making sense through dependency graphs

```
future<void> binary_expression::analyze(analysis_context & ctx)
{
    return when_all(
        lhs->analyze(ctx),
        rhs->analyze(ctx)
    )
    .then([&]{
        return resolve_overload(ctx, lhs.get(), rhs.get(), op);
    })
    .then([&](std::unique_ptr<expression> call){
        call_expr = std::move(call);
        return call_expr->analyze(ctx);
    })
    .then([&]{
        set_type(call_expr->get_type());
    });
}
```

Making sense through dependency graphs

```
function factorial(i : int) -> int
{
    if (i == 1) { return 1; }
    return i * factorial(i - 1);
}
```

Making sense through dependency graphs

```
function factorial(i : int) -> int
{
    if (i == 1) { return 1; }
    return i * factorial(i - 1);
}
```

- to analyze `factorial`, we need to analyze its signature and its body

Making sense through dependency graphs

```
function factorial(i : int) -> int
{
    if (i == 1) { return 1; }
    return i * factorial(i - 1);
}
```

- to analyze `factorial`, we need to analyze its signature and its body
- to analyze the call to `factorial`, we need to know what its signature is

Making sense through dependency graphs

```
function factorial(i : int)
{
    if (i == 1) { return 1; }
    return i * factorial(i - 1);
}
```

- to analyze `factorial`, we need to analyze its signature and its body
- to analyze the call to `factorial`, we need to know what its signature is

Making sense through dependency graphs

```
function factorial(i : int)
{
    if (i == 1) { return 1; }
    return i * factorial(i - 1);
}
```

- to analyze `factorial`, we need to analyze its signature and its body
- to analyze the call to `factorial`, we need to know what its signature is
- now we to solve a system of equations:
  return_type_of(factorial) == type_of(1)
  return_type_of(factorial) == type_of(i * factorial(i - 1))

Making sense through dependency graphs

```
future<void> function_definition::analyze(analysis_context & ctx)
{
    return when_all(fmap(parameters, [&](auto && param){
        return param->analyze(ctx);
    }))
```

Making sense through dependency graphs

```
future<void> function_definition::analyze(analysis_context & ctx)
{
    return when_all(fmap(parameters, [&](auto && param){
        return param->analyze(ctx);
    }))
        .then([&]{
            return return_type->analyze(ctx);
        })
```

Making sense through dependency graphs

```
future<void> function_definition::analyze(analysis_context & ctx)
{
    return when_all(fmap(parameters, [&](auto && param){
        return param->analyze(ctx);
    }))
        .then([&]{
            return return_type->analyze(ctx);
        })
        .then([&]{
            return body->analyze(ctx);
        });
}
```

Outline

1. Structure of compilers and interpreters
2. Creating ASTs
3. Processing ASTs
4. What I tried in an interpreter
5. What I tried in a compiler
6. Real world

Clang

```
Decl *Parser::ParseExportDeclaration() {
  assert(Tok.is(tok::kw_export));
  SourceLocation ExportLoc = ConsumeToken();
  ParseScope ExportScope(this, Scope::DeclScope);
  Decl *ExportDecl = Actions.ActOnStartExportDecl(getCurScope(), ExportLoc,
      Tok.is(tok::l_brace) ? Tok.getLocation() : SourceLocation());
  if (Tok.isNot(tok::l_brace)) {
    // FIXME: Factor out a ParseExternalDeclarationWithAttrs.
    // ... SNIP ...
    ParseExternalDeclaration(Attrs);
    return Actions.ActOnFinishExportDecl(getCurScope(), ExportDecl,
                                         SourceLocation());
  }
  // ... SNIP ...
  while (/* ... SNIP ... */) {
      // ... SNIP ...
      ParseExternalDeclaration(Attrs);
  }
  // ... SNIP ...
  return Actions.ActOnFinishExportDecl(getCurScope(), ExportDecl,
                                       T.getCloseLocation());
}
```

Clang

```
/// We have parsed the start of an export declaration, including the '{'
/// (if present).
Decl *Sema::ActOnStartExportDecl(Scope *S, SourceLocation ExportLoc,
                                 SourceLocation LBraceLoc) {
  ExportDecl *D = ExportDecl::Create(Context, CurContext, ExportLoc);
  // C++ Modules TS draft:
  //   An export-declaration shall appear in the purview of a module other than
  //   the global module.
  if (ModuleScopes.empty() || !ModuleScopes.back().ModuleInterface)
    Diag(ExportLoc, diag::err_export_not_in_module_interface);
  //   An export-declaration [...] shall not contain more than one
  //   export keyword.
  //
  // The intent here is that an export-declaration cannot appear within another
  // export-declaration.
  if (D->isExported())
    Diag(ExportLoc, diag::err_export_within_export);
  CurContext->addDecl(D);
  PushDeclContext(S, D);
  D->setModuleOwnershipKind(Decl::ModuleOwnershipKind::VisibleWhenImported);
  return D;
}
```

Clang

```
/// Complete the definition of an export declaration.
Decl *Sema::ActOnFinishExportDecl(Scope *S, Decl *D,
                                  SourceLocation RBraceLoc) {
  auto *ED = cast<ExportDecl>(D);
  if (RBraceLoc.isValid())
    ED->setRBraceLoc(RBraceLoc);
  // FIXME: Diagnose export of internal-linkage declaration (including
  // anonymous namespace).
  PopDeclContext();
  return D;
}
```

From parsing to sema: making sense of syntax trees

Michał Dominiak
Nokia Networks
griwes@griwes.info