Presentation of TC-2

Assistants 2009

May 6, 2014

Presentation of TC-2

Overview of the tarball

The tree structure of TC-2

- It is the same structure as TC-1.
- Only the ‘`src/ast`’ directory has been added.

The tree structure of TC-2

- It is the same structure as TC-1.
- Only the '`src/ast`' directory has been added.

misc::Error

- It is a class used to centralize all error reporting.
- One global error handler of that class lays in ‘`src/common.cc`’.
- Used like a stream (redefines `operator<<`)
  - We can use it for printing any object (if it implements `operator<<`).
  - It defines several manipulators for setting error states.

```
error_ << misc::Error::scan
<< e.location_get ()
<< ": unexpected end of file " << std::endl;
```

- `exit` and `exit_on_error` throw an Error object when needed, caught in ‘`tc.cc`’.

misc::Error

- It is a class used to centralize all error reporting.
- One global error handler of that class lays in '`src/common.cc`'.
- Used like a stream (redefines `operator<<`)
  - We can use it for printing any object (if it implements `operator<<`).
  - It defines several manipulators for setting error state.

```
error_ << misc::Error::scan
<< e.location_get ()
<< ": unexpected end of file " << std::endl;
```

- `exit` and `exit_on_error` throw an Error object when needed, caught in '`tc.cc`'.

misc::Error

- It is a class used to centralize all error reporting.
- One global error handler of that class lays in '`src/common.cc`'.
- Used like a stream (redefines `operator<<`)
  - We can use it for printing any object (if it implements `operator<<`).
  - It defines several manipulators for setting error states.

```
error_ << misc::Error::scan
<< e.location_get ()
<< ": unexpected end of file " << std::endl;
```

- `exit` and `exit_on_error` throw an Error object when needed, caught in '`tc.cc`'.

misc::Error

- It is a class used to centralize all error reporting.
- One global error handler of that class lays in '`src/common.cc`'.
- Used like a stream (redefines `operator`<<)
  - We can use it for printing any object (if it implements `operator`<<).
  - It defines several manipulators for setting error states.

```
error_ << misc::Error::scan
<< e.location_get ()
<< ": unexpected end of file " << std::endl;
```

- `exit` and `exit_on_error` throw an Error object when needed, caught in '`tc.cc`'.

misc::Error

- It is a class used to centralize all error reporting.
- One global error handler of that class lays in '`src/common.cc`'.
- Used like a stream (redefines `operator<<`)
  - We can use it for printing any object (if it implements `operator<<`).
  - It defines several manipulators for setting error states.

```
error_ << misc::Error::scan
<< e.location_get ()
<< ": unexpected end of file " << std::endl;
```

- `exit` and `exit_on_error` throw an Error object when needed, caught in '`tc.cc`'.

misc::Error

- It is a class used to centralize all error reporting.
- One global error handler of that class lays in '`src/common.cc`'.
- Used like a stream (redefines `operator<<`)
  - We can use it for printing any object (if it implements `operator<<`).
  - It defines several manipulators for setting error states.

```
 error_ << misc::Error::scan
<< e.location_get ()
<< ": unexpected end of file " << std::endl;
```

- `exit` and `exit_on_error` throw an Error object when needed, caught in '`tc.cc`'.

misc::Error

- It is a class used to centralize all error reporting.
- One global error handler of that class lays in ‘`src/common.cc`’.
- Used like a stream (redefines `operator<<`)
  - We can use it for printing any object (if it implements `operator<<`).
  - It defines several manipulators for setting error states.

```
 error_ << misc::Error::scan
<< e.location_get ()
<< ": unexpected end of file " << std::endl;
```

- `exit` and `exit_on_error` throw an Error object when needed, caught in ‘`tc.cc`’.

Introduction of _main

- The body of a Tiger program is outside any function
- This will require a specific handling in the rest of the compiler
- To simplify the compiler, the `_main` function (entry point) is introduced after the initial parsing via an AST transformation

Introduction of _main

- The body of a Tiger program is outside any function
- This will require a specific handling in the rest of the compiler
- To simplify the compiler, the `_main` function (entry point) is introduced after the initial parsing via an AST transformation

Introduction of _main

- The body of a Tiger program is outside any function
- This will require a specific handling in the rest of the compiler
- To simplify the compiler, the `_main` function (entry point) is introduced after the initial parsing via an AST transformation

Code to write

‘src/parse/parsetiger.yy’

- Implement error recovery using the error token.
- Chunks.
- Create AST nodes.

‘src/parse/parsetiger.yy’

- Implement error recovery using the error token.
- Chunks.
- Create AST nodes.

‘src/parse/parsetiger.yy’

- Implement error recovery using the error token.
- Chunks.
- Create AST nodes.

parsetiger.yy

Parser enhancement: GLR

- Upgrade your parser from LALR(1) to GLR
- Difficult reduce/reduce conflicts will disappear without grammar massaging
- Add `%skeleton "glr.cc"`, `%glr-parser`, `%expect`, `%expect-rr`

Parser enhancement: GLR

- Upgrade your parser from LALR(1) to GLR
- Difficult reduce/reduce conflicts will disappear without grammar massaging
- Add `%skeleton "glr.cc"`, `%glr-parser`, `%expect`, `%expect-rr`

Parser enhancement: GLR

- Upgrade your parser from LALR(1) to GLR
- Difficult reduce/reduce conflicts will disappear without grammar massaging
- Add `%skeleton "glr.cc"`, `%glr-parser`, `%expect`, `%expect-rr`

The error token (extract from the Bison documentation)

- You can define how to recover from a syntax error by writing rules to recognize the special token ‘error’.
- This is a terminal symbol that is always defined (you need not declare it) and reserved for error handling.
- The Bison parser generates an ‘error’ token whenever a syntax error happens; if you have provided a rule to recognize this token in the current context, the parse can continue.

The error token (extract from the Bison documentation)

- You can define how to recover from a syntax error by writing rules to recognize the special token ‘error’.
- This is a terminal symbol that is always defined (you need not declare it) and reserved for error handling.
- The Bison parser generates an ‘error’ token whenever a syntax error happens; if you have provided a rule to recognize this token in the current context, the parse can continue.

The error token (extract from the Bison documentation)

- You can define how to recover from a syntax error by writing rules to recognize the special token ‘error’.
- This is a terminal symbol that is always defined (you need not declare it) and reserved for error handling.
- The Bison parser generates an ‘error’ token whenever a syntax error happens; if you have provided a rule to recognize this token in the current context, the parse can continue.

Declarations problems

- ```
  function foo() = bar()
  function bar() = foo()
  ```
- Problem: `foo()` does not know `bar()`.
- Swapping declarations does not solve the problem.
- Several solutions:
  - Introducing forward declarations (C/C++).
  - Introducing simultaneous declarations (Caml/Ocaml).
- The problem arises only for types and functions (not for variables).
  ```

Declarations problems

- ```
  function foo() = bar()
  function bar() = foo()
  ```
- Problem: `foo()` does not know `bar()`.
- Swapping declarations does not solve the problem.
- Several solutions:
  - Introducing forward declarations (C/C++).
  - Introducing simultaneous declarations (Caml/Ocaml).
- The problem arises only for types and functions (not for variables).
  ```

Declarations problems

- ```
  function foo() = bar()
  function bar() = foo()
  ```
- Problem: `foo()` does not know `bar()`.
- Swapping declarations does not solve the problem.
- Several solutions:
  - Introducing forward declarations (C/C++).
  - Introducing simultaneous declarations (Caml/Ocaml only).
- The problem arises only for types and functions (not for variables).
  ```

Declarations problems

- ```
  function foo() = bar()
  function bar() = foo()
  ```
- Problem: `foo()` does not know `bar()`.
- Swapping declarations does not solve the problem.
- Several solutions:
  - Introducing forward declarations (C/C++).
  - Introducing simultaneous declarations (Tiger/Caml).
- The problem arises only for types and functions (not for variables).
  ```

Declarations problems

- ```
  function foo() = bar()
  function bar() = foo()
  ```
- Problem: `foo()` does not know `bar()`.
- Swapping declarations does not solve the problem.
- Several solutions:
  - Introducing forward declarations (C/C++).
  - Introducing simultaneous declarations (Tiger/Caml).
- The problem arises only for types and functions (not for variables).
  ```

Declarations problems

- ```
  function foo() = bar()
  function bar() = foo()
  ```
- Problem: `foo()` does not know `bar()`.
- Swapping declarations does not solve the problem.
- Several solutions:
  - Introducing forward declarations (C/C++).
  - Introducing simultaneous declarations (Tiger/Caml).
- The problem arises only for types and functions (not for variables).
  ```

Declarations problems

- ```
  function foo() = bar()
  function bar() = foo()
  ```
- Problem: `foo()` does not know `bar()`.
- Swapping declarations does not solve the problem.
- Several solutions:
  - Introducing forward declarations (C/C++).
  - Introducing simultaneous declarations (Tiger/Caml).
- The problem arises only for types and functions (not for variables).
  ```

What is a chunk?

- It is a bunch of declarations of the same type (for types and functions). Each declaration of variable uses a chunk. For example :

```
let
/* declarations of a and b are not in the same chunk. */
  var a := 1
  var b := 2
/* declarations of foo and bar are in another chunk. */
  function foo () : int = 1
  function bar () : int = 2
/* declaration of c is in a fourth one. */
  var c := 3
/* declarations of tree and graph are in another chunk. */
  type tree = {graph : g, tree : fg, tree : fd}
  type graph = {int index, tree : tree}
in
  0
end
```

Why chunks?

- "Chunks" is a name only used in EPITA.
- It is useful for interdependant functions or types.
- All the entities in a chunk are declared simultaneously. So, the following example is illegal:

```
let
  type tree = {graph : g, tree : fg, tree : fd}
  var a := 1
  type graph = {int index, tree : tree}
in
  0
end
```

Why chunks?

- "Chunks" is a name only used in EPITA.
- It is useful for interdependant functions or types.
- All the entities in a chunk are declared simultaneously. So, the following example is illegal:

```
let
  type tree = {graph : g, tree : fg, tree : fd}
  var a := 1
  type graph = {int index, tree : tree}
in
  0
end
```

Why chunks?

- "Chunks" is a name only used in EPITA.
- It is useful for interdependant functions or types.
- All the entities in a chunk are declared simultaneously. So, the following example is illegal:

```
let
  type tree = {graph : g, tree : fg, tree : fd}
  var a := 1
  type graph = {int index, tree : tree}
in
  0
end
```

How are the chunks implemented?

- A list of `Decs` is used in the `LetExp` class. It is declared in the 'ast/decs-list' class:

```
/// DecsList.
class DecsList: public Ast
{
public:
  typedef std::list<Decs*> decs_type;
```

Creation of the ast nodes

- You have to implement the creation of the AST nodes in the Bison file (by creating the corresponding classes)
- Beware of default action when specifying no code: `$$=$1`
- Pay attention to memory leaks!

Creation of the ast nodes

- You have to implement the creation of the AST nodes in the Bison file (by creating the corresponding classes)
- Beware of default action when specifying no code: `$$=$1`
- Pay attention to memory leaks!

Creation of the ast nodes

- You have to implement the creation of the AST nodes in the Bison file (by creating the corresponding classes)
- Beware of default action when specifying no code: `$$=$1`
- Pay attention to memory leaks!

The ast

‘src/ast’

- The different Visitors.
- The different Nodes.

‘src/ast’

- The different Visitors.
- The different Nodes.

The visitor

- Encapsulate an operation you want to perform on a data structure.
- Adding new operations without changing the classes of the visited elements.
- Decouple the classes and the algorithms used.

The visitor

- Encapsulate an operation you want to perform on a data structure.
- Adding new operations without changing the classes of the visited elements.
- Decouple the classes and the algorithms used.

The visitor

- Encapsulate an operation you want to perform on a data structure.
- Adding new operations without changing the classes of the visited elements.
- Decouple the classes and the algorithms used.

The visitor

- A node “accepts” a Visitor passing itself in. The visitor then executes the algorithm.
- This is "Double Dispatching".
- Call depends on the Visitor and on the Host (data structure node),

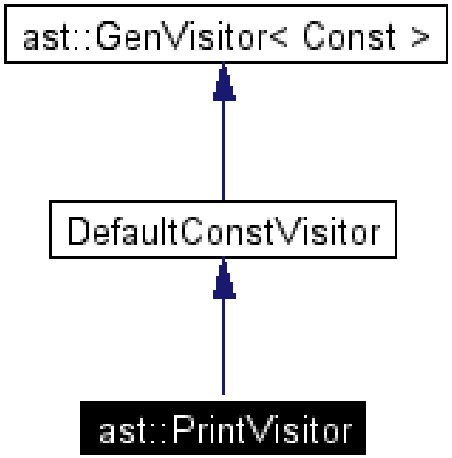
The visitor

- A node “accepts” a Visitor passing itself in. The visitor then executes the algorithm.
- This is "Double Dispatching".
- Call depends on the Visitor and on the Host (data structure node),

The visitor

- A node “accepts” a Visitor passing itself in. The visitor then executes the algorithm.
- This is "Double Dispatching".
- Call depends on the Visitor and on the Host (data structure node),

The Visitor Hierarchy



- 2 main sorts of visitors: const and non const.

```
/// Shorthand for a const visitor.
typedef GenDefaultVisitor<misc::constify_traits>
          DefaultConstVisitor;
/// Shorthand for a non const visitor.
typedef GenDefaultVisitor<misc::id_traits>
          DefaultVisitor;
```

The PrettyPrinter

- Think about using `misc::indent` to indent your ast printing.
- Instead of using the accept method on each node of the AST, use operator<<, which is cleaner and easier to read, as defined in 'ast/libast.cc'.

```
// Print the TREE on OSTR.
std::ostream&
operator<< (std::ostream& ostr, const Ast& tree)
{
  PrettyPrinter pv (ostr);
  tree.accept (pv);
  return ostr;
}
```

The PrettyPrinter

- Think about using `misc::indent` to indent your ast printing.
- Instead of using the `accept` method on each node of the AST, use `operator<<`, which is cleaner and easier to read, as defined in 'ast/libast.cc'.

```
// Print the TREE on OSTR.
std::ostream&
operator<< (std::ostream& ostr, const Ast& tree)
{
  PrettyPrinter pv (ostr);
  tree.accept (pv);
  return ostr;
}
```

xalloc [1]

- When user-defined I/O operators are being written, it is often desirable to have formatting flags specific to these operators, probably set by using a corresponding manipulator.
- The stream objects support this by providing a mechanism to associate data with a stream. This mechanism can be used to associate corresponding data (for example, using a manipulator), and later retrieve the data.
- The class `ios_base` defines the two functions `iword()` and `pword()`, each taking an int argument as the index, to access a specific `long&` or `void*&` respectively.

xalloc [1]

- When user-defined I/O operators are being written, it is often desirable to have formatting flags specific to these operators, probably set by using a corresponding manipulator.
- The stream objects support this by providing a mechanism to associate data with a stream. This mechanism can be used to associate corresponding data (for example, using a manipulator), and later retrieve the data.
- The class `ios_base` defines the two functions `iword()` and `pword()`, each taking an int argument as the index, to access a specific `long&` or `void*&` respectively.

xalloc [1]

- When user-defined I/O operators are being written, it is often desirable to have formatting flags specific to these operators, probably set by using a corresponding manipulator.
- The stream objects support this by providing a mechanism to associate data with a stream. This mechanism can be used to associate corresponding data (for example, using a manipulator), and later retrieve the data.
- The class `ios_base` defines the two functions `iword()` and `pword()`, each taking an int argument as the index, to access a specific `long&` or `void*&` respectively.

xalloc [1]

- The idea is that `iword()` and `pword()` access `long` or `void*` objects in an array of arbitrary size stored with a stream object. Formatting flags to be stored for a stream are then placed at the same index for all streams.
- The static member function `xalloc()` of the class `ios_base` is used to obtain an index that is not yet used for this purpose.
- Initially, the objects accessed with `iword()` or `pword()` are set to 0. This value can be used to represent the default formatting or to indicate that the corresponding data was not yet accessed.

xalloc [1]

- The idea is that `iword()` and `pword()` access `long` or `void*` objects in an array of arbitrary size stored with a stream object. Formatting flags to be stored for a stream are then placed at the same index for all streams.
- The static member function `xalloc()` of the class `ios_base` is used to obtain an index that is not yet used for this purpose.
- Initially, the objects accessed with `iword()` or `pword()` are set to 0. This value can be used to represent the default formatting or to indicate that the corresponding data was not yet accessed.

xalloc [1]

- The idea is that `iword()` and `pword()` access `long` or `void*` objects in an array of arbitrary size stored with a stream object. Formatting flags to be stored for a stream are then placed at the same index for all streams.
- The static member function `xalloc()` of the class `ios_base` is used to obtain an index that is not yet used for this purpose.
- Initially, the objects accessed with `iword()` or `pword()` are set to 0. This value can be used to represent the default formatting or to indicate that the corresponding data was not yet accessed.

xalloc: example

```
// get index for new ostream data
static const int iword_index = std::ios_base::xalloc();

// define manipulator that sets this data
std::ostream& fraction_spaces (std::ostream& o)
{
    o.iword(iword_index) = true;
    return o;
}
std::ostream& operator<< (std::ostream& o,
                          const Fraction& f)
{
    if (o.iword(iword_index))
        o << f.numerator() << " / " << f.denominator();
    else
        o << f.numerator() << "/" << f.denominator();
     return o;
}
```

Improvements

& and | desugaring

- Desugar the & and | operators using `if` statements
- 2 solutions:
  - Instantiate AST nodes by hand (`IfExp`, etc.)
  - Use the power of the parser and Tweast to desugar programs on-the-fly

& and | desugaring

- Desugar the & and | operators using `if` statements
- 2 solutions:
  - Instantiate AST nodes by hand (`IfExp`, etc.)
  - Use the power of the parser and Tweast to desugar in concrete syntax

& and | desugaring

- Desugar the & and | operators using `if` statements
- 2 solutions:
  - Instantiate AST nodes by hand (`IfExp`, etc.)
  - Use the power of the parser and Tweast to desugar in concrete syntax

& and | desugaring

- Desugar the & and | operators using `if` statements
- 2 solutions:
  - Instantiate AST nodes by hand (`IfExp`, etc.)
  - Use the power of the parser and Tweast to desugar in concrete syntax

Tweast

- The parser is very good at creating AST
- Why should we do it by hand?
- The Tweast mixes Tiger code (strings) with already constructed AST
- When parsing a Tweast, strings are parsed again, but AST are just plugged in “holes”

Tweast

- The parser is very good at creating AST
- Why should we do it by hand?
- The Tweast mixes Tiger code (strings) with already constructed AST
- When parsing a Tweast, strings are parsed again, but AST are just plugged in “holes”

Tweast

- The parser is very good at creating AST
- Why should we do it by hand?
- The Tweast mixes Tiger code (strings) with already constructed AST
- When parsing a Tweast, strings are parsed again, but AST are just plugged in “holes”

Tweast

- The parser is very good at creating AST
- Why should we do it by hand?
- The Tweast mixes Tiger code (strings) with already constructed AST
- When parsing a Tweast, strings are parsed again, but AST are just plugged in “holes”

Adding an entry point (1/2)

- A good example of the use of concrete syntax with Tweast is _main
- **Goal** Apply this transformation:
  $program \mapsto$ **function** _main () = (*program*; ())
  where *program* is an exp

Adding an entry point (1/2)

- A good example of the use of concrete syntax with Tweast is _main
- **Goal** Apply this transformation:
  $program \mapsto$ **function** _main () = (*program*; ())
  where *program* is an exp

Adding an entry point (2/2)

- Using abstract syntax:

```
ast::Exp** exp = boost::get<ast::Exp*> (&tree);
ast::Location loc = exp->location_get ();
ast::exps_type* exps = new exps_type ();
exps->push_back (*exp);
exps->push_back (new ast::SeqExp ());
ast::SeqExp* body = new ast::SeqExp (loc, exps);
ast::FunctionDecs* fundecs = new ast::FunctionDecs ();
fundecs.push_back
  (ast::FunctionDec (loc, "_main", new ast::VarDecs (loc),
                     0, body));
res = new ast::DecsList ()
res.push_front (fundecs);
```

- Using concrete syntax:

```
ast::Exp** exp = boost::get<ast::Exp*> (&tree);
res = tp.enable_extensions ().parse (Tweast () <<
  "function _main () = (" << *exp << "; ())");
```

- Which one do you prefer?

Adding an entry point (2/2)

- Using abstract syntax:

```
ast::Exp** exp = boost::get<ast::Exp*> (&tree);
ast::Location loc = exp->location_get ();
ast::exps_type* exps = new exps_type ();
exps->push_back (*exp);
exps->push_back (new ast::SeqExp ());
ast::SeqExp* body = new ast::SeqExp (loc, exps);
ast::FunctionDecs* fundecs = new ast::FunctionDecs ();
fundecs.push_back
  (ast::FunctionDec (loc, "_main", new ast::VarDecs (loc),
                     0, body));
res = new ast::DecsList ()
res.push_front (fundecs);
```

- Using concrete syntax:

```
ast::Exp** exp = boost::get<ast::Exp*> (&tree);
res = tp.enable_extensions ().parse (Tweast () <<
  "function _main () = (" << *exp << "; ())");
```

- Which one do you prefer?

Adding an entry point (2/2)

- Using abstract syntax:

```
ast::Exp** exp = boost::get<ast::Exp*> (&tree);
ast::Location loc = exp->location_get ();
ast::exps_type* exps = new exps_type ();
exps->push_back (*exp);
exps->push_back (new ast::SeqExp ());
ast::SeqExp* body = new ast::SeqExp (loc, exps);
ast::FunctionDecs* fundecs = new ast::FunctionDecs ();
fundecs.push_back
  (ast::FunctionDec (loc, "_main", new ast::VarDecs (loc),
                     0, body));
res = new ast::DecsList ()
res.push_front (fundecs);
```

- Using concrete syntax:

```
ast::Exp** exp = boost::get<ast::Exp*> (&tree);
res = tp.enable_extensions ().parse (Tweast () <<
  "function _main () = (" << *exp << "; ())");
```

- Which one do you prefer?

Bibliography I


Nicolai M. Josuttis.
The C++ standard library: A tutorial and reference, 1999.