Presentation of TC-1

Assistants 2009

May 6, 2014

Presentation of TC-1

Overview of the tarball

The tree structure of TC-1

- lib/misc : Contains many tools used in the whole project:
  - unique : Contains the class Unique that implements the design pattern Flyweight.
  - symbol : Contains the class Symbol which inherits from **misc:unique<std::string>**. This class maps any string to a unique reference.
  - variant : Contains the class Variant which inherits from **boost::variant**.
- src/parse : Contains the flex and bison files (parsetiger.yy, scantiger.ll).
- src/task : Tiger is cut in “tasks” (parse, building of the ast, type-checking, etc.). The tasks manager is contained in that directory. This module is based on the design pattern Command

The tree structure of TC-1

- lib/misc : Contains many tools used in the whole project:
  - unique : Contains the class Unique that implements the design pattern Flyweight.
  - symbol : Contains the class Symbol which inherits from **misc::unique<std::string>**. This class maps any string to a unique reference.
  - variant : Contains the class Variant which inherits from **boost::variant**.
- src/parse : Contains the flex and bison files (parsetiger.yy, scantiger.ll).
- src/task : Tiger is cut in "tasks" (parse, building of the ast, type-checking, etc.). The tasks manager is contained in that directory. This module is based on the design pattern Command

The tree structure of TC-1

- lib/misc : Contains many tools used in the whole project:
  - unique : Contains the class Unique that implements the design pattern Flyweight.
  - symbol : Contains the class Symbol which inherits from **misc:unique<std::string>**. This class maps any string to a unique reference.
  - variant : Contains the class Variant which inherits from **boost::variant**.
- src/parse : Contains the flex and bison files (parsetiger.yy, scantiger.ll).
- src/task : Tiger is cut in “tasks” (parse, building of the ast, type-checking, etc.). The tasks manager is contained in that directory. This module is based on the design pattern Command

The tree structure of TC-1

- lib/misc : Contains many tools used in the whole project:
  - unique : Contains the class Unique that implements the design pattern Flyweight.
  - symbol : Contains the class Symbol which inherits from **misc:unique<std::string>**. This class maps any string to a unique reference.
  - variant : Contains the class Variant which inherits from **boost::variant**.
- src/parse : Contains the flex and bison files (parsetiger.yy, scantiger.ll).
- src/task : Tiger is cut in "tasks" (parse, building of the ast, type-checking, etc.). The tasks manager is contained in that directory. This module is based on the design pattern Command

The tree structure of TC-1

- lib/misc : Contains many tools used in the whole project:
  - unique : Contains the class Unique that implements the design pattern Flyweight.
  - symbol : Contains the class Symbol which inherits from **misc:unique<std::string>**. This class maps any string to a unique reference.
  - variant : Contains the class Variant which inherits from **boost::variant**.
- src/parse : Contains the flex and bison files (parsetiger.yy, scantiger.ll).
- src/task : Tiger is cut in “tasks” (parse, building of the ast, type-checking, etc.). The tasks manager is contained in that directory. This module is based on the design pattern Command

The tree structure of TC-1

- lib/misc : Contains many tools used in the whole project:
  - unique : Contains the class Unique that implements the design pattern Flyweight.
  - symbol : Contains the class Symbol which inherits from **misc:unique<std::string>**. This class maps any string to a unique reference.
  - variant : Contains the class Variant which inherits from **boost::variant**.
- src/parse : Contains the flex and bison files (parsetiger.yy, scantiger.ll).
- src/task : Tiger is cut in “tasks” (parse, building of the ast, type-checking, etc.). The tasks manager is contained in that directory. This module is based on the design pattern Command

The tree structure of TC-1

- lib/misc : Contains many tools used in the whole project:
  - unique : Contains the class Unique that implements the design pattern Flyweight.
  - symbol : Contains the class Symbol which inherits from **misc:unique<std::string>**. This class maps any string to a unique reference.
  - variant : Contains the class Variant which inherits from **boost::variant**.
- src/parse : Contains the flex and bison files (parsetiger.yy, scantiger.ll).
- src/task : Tiger is cut in “tasks” (parse, building of the ast, type-checking, etc.). The tasks manager is contained in that directory. This module is based on the design pattern Command

Code to write

- autotools : Customize the main files: configure.ac, Makefile.am, AUTHORS.
- src/parse/scantiger.ll : Complete the scanner.
- src/parse/parsetiger.yy : Complete the parser.
- src/parse/tiger-parser.cc : Complete the driver.
- lib/misc/unique.* : Complete the Flyweight design pattern in Unique class.
- lib/misc/symbol.* : Complete the Symbol class which inherits from Unique.
- lib/misc/variant.* : Complete the Variant class which inherits from boost::variant.
- More details on assignments.

Code to write

- autotools : Customize the main files: configure.ac, Makefile.am, AUTHORS.
- src/parse/scantiger.ll : Complete the scanner.
- src/parse/parsetiger.yy : Complete the parser.
- src/parse/tiger-parser.cc : Complete the driver.
- lib/misc/unique.* : Complete the Flyweight design pattern in Unique class.
- lib/misc/symbol.* : Complete the Symbol class which inherits from Unique.
- lib/misc/variant.* : Complete the Variant class which inherits from boost::variant.
- More details on assignments.

Code to write

- autotools : Customize the main files: configure.ac, Makefile.am, AUTHORS.
- src/parse/scantiger.ll : Complete the scanner.
- src/parse/parsetiger.yy : Complete the parser.
- src/parse/tiger-parser.cc : Complete the driver.
- lib/misc/unique.* : Complete the Flyweight design pattern in Unique class.
- lib/misc/symbol.* : Complete the Symbol class which inherits from Unique.
- lib/misc/variant.* : Complete the Variant class which inherits from boost::variant.
- More details on assignments.

Code to write

- autotools : Customize the main files: configure.ac, Makefile.am, AUTHORS.
- src/parse/scantiger.ll : Complete the scanner.
- src/parse/parsetiger.yy : Complete the parser.
- src/parse/tiger-parser.cc : Complete the driver.
- lib/misc/unique.* : Complete the Flyweight design pattern in Unique class.
- lib/misc/symbol.* : Complete the Symbol class which inherits from Unique.
- lib/misc/variant.* : Complete the Variant class which inherits from boost::variant.
- More details on assignments.

Code to write

- autotools : Customize the main files: configure.ac, Makefile.am, AUTHORS.
- src/parse/scantiger.ll : Complete the scanner.
- src/parse/parsetiger.yy : Complete the parser.
- src/parse/tiger-parser.cc : Complete the driver.
- lib/misc/unique.* : Complete the Flyweight design pattern in Unique class.
- lib/misc/symbol.* : Complete the Symbol class which inherits from Unique.
- lib/misc/variant.* : Complete the Variant class which inherits from boost::variant.
- More details on assignments.

Code to write

- autotools : Customize the main files: configure.ac, Makefile.am, AUTHORS.
- src/parse/scantiger.ll : Complete the scanner.
- src/parse/parsetiger.yy : Complete the parser.
- src/parse/tiger-parser.cc : Complete the driver.
- lib/misc/unique.* : Complete the Flyweight design pattern in Unique class.
- lib/misc/symbol.* : Complete the Symbol class which inherits from Unique.
- lib/misc/variant.* : Complete the Variant class which inherits from boost::variant.
- More details on assignments.

Code to write

- autotools : Customize the main files: configure.ac, Makefile.am, AUTHORS.
- src/parse/scantiger.ll : Complete the scanner.
- src/parse/parsetiger.yy : Complete the parser.
- src/parse/tiger-parser.cc : Complete the driver.
- lib/misc/unique.* : Complete the Flyweight design pattern in Unique class.
- lib/misc/symbol.* : Complete the Symbol class which inherits from Unique.
- lib/misc/variant.* : Complete the Variant class which inherits from boost::variant.
- More details on assignments.

Code to write

- autotools : Customize the main files: configure.ac, Makefile.am, AUTHORS.
- src/parse/scantiger.ll : Complete the scanner.
- src/parse/parsetiger.yy : Complete the parser.
- src/parse/tiger-parser.cc : Complete the driver.
- lib/misc/unique.* : Complete the Flyweight design pattern in Unique class.
- lib/misc/symbol.* : Complete the Symbol class which inherits from Unique.
- lib/misc/variant.* : Complete the Variant class which inherits from boost::variant.
- More details on assignments.

The GNU Build System

The gnu Build System [2]

- Customize the configure.ac
- Bootstrap the package
- Compile:

-
  ```
  $ mkdir -m 700 _build
  $ cd _build
  $ ../configure --with-boost=/usr/local/
  $ make
  ```
- Deliver your tarball: make distcheck

The gnu Build System [2]

- Customize the configure.ac
- Bootstrap the package
- Compile:

-
  ```
  $ mkdir -m 700 _build
  $ cd _build
  $ ../configure --with-boost=/usr/local/
  $ make
  ```
- Deliver your tarball: make distcheck

The gnu Build System [2]

- Customize the configure.ac
- Bootstrap the package
- Compile:

-
  ```
  $ mkdir -m 700 _build
  $ cd _build
  $ ../configure --with-boost=/usr/local/
  $ make
  ```
- Deliver your tarball: make distcheck

The gnu Build System [2]

- Customize the configure.ac
- Bootstrap the package
- Compile:
-
```
$ mkdir -m 700 _build
$ cd _build
$ ../configure --with-boost=/usr/local/
$ make
```
- Deliver your tarball: make distcheck

The gnu Build System [2]

- Customize the configure.ac
- Bootstrap the package
- Compile:

-
  ```
  $ mkdir -m 700 _build
  $ cd _build
  $ ../configure --with-boost=/usr/local/
  $ make
  ```
- Deliver your tarball: make distcheck

The use of CONFIG_SITE

- The **configure** script needs options when compiling on the PIE
- The environment variable CONFIG_SITE allows to specify a shell script that will be used by **configure**
- In particular, this script allows to set the path to boost or the C++ compiler to use

The use of CONFIG_SITE

- The **configure** script needs options when compiling on the PIE
- The environment variable CONFIG_SITE allows to specify a shell script that will be used by **configure**
- In particular, this script allows to set the path to `boost` or the C++ compiler to use

The use of CONFIG_SITE

- The **configure** script needs options when compiling on the PIE
- The environment variable CONFIG_SITE allows to specify a shell script that will be used by **configure**
- In particular, this script allows to set the path to `boost` or the C++ compiler to use

Example of CONFIG_SITE

-
  ```
  $ cat ~/config.site
  MAKE=gmake
  CC=/usr/local/bin/gcc41
  CXX=/usr/local/bin/g++41
  BISON=/u/all/acu/public/bin/bison
  FLEX=flex
  $ export CONFIG_SITE=~/config.site
  ```

Variant types

Unions in C++

- Unions in C++ are quite useless.
- Can be used only with POD (Plain Old Data).
- How to initialize v?

```
union Value
{
  ast::IntExp ival;
  ast::StringExp str;
};

int main()
{
  Value v;
}
```

- The compiler cannot know which constructor to call.

Unions in C++

- Using pointer to objects “solves” the issue.
- Boost addresses the issue with **Variant**.
- Boost is a C++ library like the STL. (`http://www.boost.org`)
- It provides a lot of useful classes.
- Simple example of **Variant** use:

```
boost::variant<int, std::string> v;

// Print "a string".
v = "a string";
std::cout << v << std::endl;

// Print 42.
v = 42;
std::cout << v << std::endl;
```

Unions in C++

- Using pointer to objects “solves” the issue.
- Boost addresses the issue with **Variant**.
- Boost is a C++ library like the STL. (`http://www.boost.org`)
- It provides a lot of useful classes.
- Simple example of **Variant** use:

```
boost::variant<int, std::string> v;

// Print "a string".
v = "a string";
std::cout << v << std::endl;

// Print 42.
v = 42;
std::cout << v << std::endl;
```

Unions in C++

- Using pointer to objects “solves” the issue.
- Boost addresses the issue with **Variant**.
- Boost is a C++ library like the STL. (http://www.boost.org)
- It provides a lot of useful classes.
- Simple example of **Variant** use:

```
boost::variant<int, std::string> v;

// Print "a string".
v = "a string";
std::cout << v << std::endl;

// Print 42.
v = 42;
std::cout << v << std::endl;
```

Unions in C++

- Using pointer to objects “solves” the issue.
- Boost addresses the issue with **Variant**.
- Boost is a C++ library like the STL. (http://www.boost.org)
- It provides a lot of useful classes.
- Simple example of **Variant** use:

```
boost::variant<int, std::string> v;

// Print "a string".
v = "a string";
std::cout << v << std::endl;

// Print 42.
v = 42;
std::cout << v << std::endl;
```

Unions in C++

- Using pointer to objects "solves" the issue.
- Boost addresses the issue with **Variant**.
- Boost is a C++ library like the STL. (http://www.boost.org)
- It provides a lot of useful classes.
- Simple example of **Variant** use:

```
boost::variant<int, std::string> v;

// Print "a string".
v = "a string";
std::cout << v << std::endl;

// Print 42.
v = 42;
std::cout << v << std::endl;
```

Using Variant: get

- **get** allows to specify the expected content type.
- Only run-time check
- When returning a pointer, **get** returns 0 if the actual value of the variant does not match the requested type:

```
boost::variant<int, std::string> v;

// value == 0.
v = "a string";
int* value = boost::get<int> (&v);
```

Using Variant: get

- **get** allows to specify the expected content type.
- Only run-time check
- When returning a pointer, **get** returns 0 if the actual value of the variant does not match the requested type:

```
boost::variant<int, std::string> v;

// value == 0.
v = "a string";
int* value = boost::get<int> (&v);
```

Using Variant: get

- **get** allows to specify the expected content type.
- Only run-time check
- When returning a pointer, **get** returns 0 if the actual value of the variant does not match the requested type:

```
boost::variant<int, std::string> v;

// value == 0.
v = "a string";
int* value = boost::get<int> (&v);
```

Using Variant: get

- When returning a reference, **get** throws a exception.

```
boost::variant<int, std::string> v;

// Throw a bad_get exception.
// Note that we do not pass ‘&v‘ but ‘v‘.
v = "a string";
int& value = boost::get<int> (v);
```

- Do you want to know more ? See **src/parse/tiger-parser** and **lib/misc/variant**

Using Variant: get

- When returning a reference, **get** throws a exception.

```
boost::variant<int, std::string> v;

// Throw a bad_get exception.
// Note that we do not pass ‘&v‘ but ‘v‘.
v = "a string";
int& value = boost::get<int> (v);
```

- Do you want to know more ? See **src/parse/tiger-parser** and **lib/misc/variant**

Templates: Declarations

The Implicit Declaration

- You are used to using explicit instantiation
- The compiler automatically creates the class or function when needed
- For example

```
template <typename T>
class List<T>
{
  //...
}

int main ()
{
  List<int> l;
}
```

The Implicit Declaration

- You are used to using explicit instantiation
- The compiler automatically creates the class or function when needed
- For example

```
template <typename T>
class List<T>
{
  //...
}

int main ()
{
  List<int> l;
}
```

The Implicit Declaration

- You are used to using explicit instantiation
- The compiler automatically creates the class or function when needed
- For example

```
template <typename T>
class List<T>
{
  //...
}

int main ()
{
  List<int> l;
}
```

The Explicit Declaration

- Templates are used both for genericity and for code factorization
- When used to factor code, some parameters' values should not be allowed.
- Hence, tell the compiler explicitely what parameters are expected. It's the Template Specialization.
- Templates are present almost everywhere in TC.

The Explicit Declaration

- Templates are used both for genericity and for code factorization
- When used to factor code, some parameters' values should not be allowed.
- Hence, tell the compiler explicitely what parameters are expected. It's the Template Specialization.
- Templates are present almost everywhere in TC.

The Explicit Declaration

- Templates are used both for genericity and for code factorization
- When used to factor code, some parameters' values should not be allowed.
- Hence, tell the compiler explicitely what parameters are expected. It's the Template Specialization.
- Templates are present almost everywhere in TC.

The Explicit Declaration

- Templates are used both for genericity and for code factorization
- When used to factor code, some parameters' values should not be allowed.
- Hence, tell the compiler explicitely what parameters are expected. It's the Template Specialization.
- Templates are present almost everywhere in TC.

The Explicit Declaration

```
template <typename T>
T
TigerParser::example ()
{
  // Implicit Declaration
}

Template <>
TigerParser::ast_type
TigerParser::example<TigerParser::ast_type> ()
{
  // Specialization
}
```

The Explicit Declaration

```
// ast_type is a typedef on a
// boost::variant<ast::Exp*, ast::DecsList*>

// So, using this instanciations are valid.
Template <>
ast::Exp*
TigerParser::example<ast::Exp*> ();

Template <>
ast::DecsList*
TigerParser::example<ast::DecsList*> ();
```

The `Unique` Class

The goal

- Implements the design pattern Flyweight.
- Relies on a std::set to do the job

The goal

- Implements the design pattern Flyweight.
- Relies on a std::set to do the job

The design pattern Flyweight [3]

- `Definition` : The Flyweight pattern provides a method to pool and share a large number of contexts. It allows a single object to be used in several contexts simultaneously.
- It looks like the pattern `Singleton` seen in the "C++-week".

The design pattern Flyweight [3]

- `Definition` : The Flyweight pattern provides a method to pool and share a large number of contexts. It allows a single object to be used in several contexts simultaneously.
- It looks like the pattern `Singleton` seen in the “C++-week”.

The Symbol Class

The goal

- Same symbol can be present numerous time.
- We want to save space and time.
- We use the `Unique` class.
- Used with the parser.

The goal

- Same symbol can be present numerous time.
- We want to save space and time.
- We use the `Unique` class.
- Used with the parser.

The goal

- Same symbol can be present numerous time.
- We want to save space and time.
- We use the `Unique` class.
- Used with the parser.

The goal

- Same symbol can be present numerous time.
- We want to save space and time.
- We use the `Unique` class.
- Used with the parser.

The import system

The prelude

- Tiger automatically imports the prelude file.
- It is mandatory for builtins.
- Default name for the prelude is “prelude.tih”.
- Default name can be changed by using “--prelude=filename”.
- It is exactly as if the program were written:

```
let
  import "prelude.tih"
in
  ...
end
```

The prelude

- Tiger automatically imports the prelude file.
- It is mandatory for builtins.
- Default name for the prelude is “prelude.tih”.
- Default name can be changed by using “--prelude=filename”.
- It is exactly as if the program were written:

```
let
  import "prelude.tih"
in
  ...
end
```

The prelude

- Tiger automatically imports the prelude file.
- It is mandatory for builtins.
- Default name for the prelude is “prelude.tih”.
- Default name can be changed by using “--prelude=filename”.
- It is exactly as if the program were written:

```
let
  import "prelude.tih"
in
  ...
end
```

The prelude

- Tiger automatically imports the prelude file.
- It is mandatory for builtins.
- Default name for the prelude is “prelude.tih”.
- Default name can be changed by using “--prelude=filename”.
- It is exactly as if the program were written:

```
let
  import "prelude.tih"
in
  ...
end
```

The prelude

- Tiger automatically imports the prelude file.
- It is mandatory for builtins.
- Default name for the prelude is “prelude.tih”.
- Default name can be changed by using “--prelude=filename”.
- It is exactly as if the program were written:

```
let
  import "prelude.tih"
in
  ...
end
```

Import processing

- Start by looking in the current directory (where the file lies).
- Then look in the include path.
- Default path contains the path PKGDATADIR or TC_PKGDATADIR environment variable if set.
- Include path can be controlled by :

  --library-prepend=DIR : prepend directory DIR to the search path.

  –library-append=DIR : append directory DIR to the search path.
- See src/parse/tasks.* and TigerParser for more infos.

Import processing

- Start by looking in the current directory (where the file lies).
- Then look in the include path.
- Default path contains the path PKGDATADIR or TC_PKGDATADIR environment variable if set.
- Include path can be controlled by :

  --library-prepend=DIR : prepend directory DIR to the search path.

  --library-append=DIR : append directory DIR to the search path.
- See src/parse/tasks.* and TigerParser for more infos.

Import processing

- Start by looking in the current directory (where the file lies).
- Then look in the include path.
- Default path contains the path PKGDATADIR or TC_PKGDATADIR environment variable if set.
- Include path can be controlled by :

  --library-prepend=DIR : prepend directory DIR to the search path.

  --library-append=DIR : append directory DIR to the search path.
- See src/parse/tasks.* and TigerParser for more infos.

Import processing

- Start by looking in the current directory (where the file lies).
- Then look in the include path.
- Default path contains the path PKGDATADIR or TC_PKGDATADIR environment variable if set.
- Include path can be controlled by :

  --library-prepend=DIR : prepend directory DIR to the search path.

  --library-append=DIR : append directory DIR to the search path.
- See src/parse/tasks.* and TigerParser for more infos.

Import processing

- Start by looking in the current directory (where the file lies).
- Then look in the include path.
- Default path contains the path PKGDATADIR or TC_PKGDATADIR environment variable if set.
- Include path can be controlled by :

--library-prepend=DIR :	prepend directory DIR to the search path.
--library-append=DIR :	append directory DIR to the search path.

- See src/parse/tasks.* and TigerParser for more infos.

Import processing

- Start by looking in the current directory (where the file lies).
- Then look in the include path.
- Default path contains the path PKGDATADIR or TC_PKGDATADIR environment variable if set.
- Include path can be controlled by :

  --library-prepend=DIR : prepend directory DIR to the search path.

  --library-append=DIR : append directory DIR to the search path.
- See src/parse/tasks.* and TigerParser for more infos.

Import processing

- Start by looking in the current directory (where the file lies).
- Then look in the include path.
- Default path contains the path PKGDATADIR or TC_PKGDATADIR environment variable if set.
- Include path can be controlled by :

  --library-prepend=DIR : prepend directory DIR to the search path.

  --library-append=DIR : append directory DIR to the search path.
- See src/parse/tasks.* and TigerParser for more infos.

The SVN server

Versioning systems

- The usage of a versioning system is mandatory.
- As usual, the SVN server from the Assistant laboratory [1] is available.

Versioning systems

- The usage of a versioning system is mandatory.
- As usual, the SVN server from the Assistant laboratory [1] is available.

Bibliography I

B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato.
Version Control With Subversion, 2004.
http://svnbook.red-bean.com.

Alexandre Duret-Lutz.
The Autotools Tutorial, 2006.
http://www-src.lip6.fr/homepages/Alexandre.Duret-Lutz/dl/autotools.pdf/.

Bibliography II

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
Design Patterns: Elements of Reusable Object-Oriented Software.
Addison-Wesley Professional Computing Series.
Addison-Wesley Publishing Company, New York, NY, 1995.