

Introduction to the Tiger Project

and Tigrou project

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

EPITA — École Pour l'Informatique et les Techniques Avancées

February 3, 2020

The Needs

- 20 years ago, EPITA asked for a long and challenging project.
- Should virtually be a *potpourri* of every subject from computer science courses taught in third year.

A (Miraculous) Solution

A compiler construction project.

The Needs

- 20 years ago, EPITA asked for a long and challenging project.
- Should virtually be a *potpourri* of every subject from computer science courses taught in third year.

A (Miraculous) Solution

A compiler construction project.

Aim

Compiler construction **as a by-product**

Complete Project

Specifications, implementation, documentation, testing, distribution.

Several iterations

6 (optionally up to 9) steps, for 3 (resp. up to 6) months.

Algorithmically challenging

Applied use of well known data structures and algorithms.

Team Management

Project conducted in group of four students.

Goals

Aim

Compiler construction as a by-product

Complete Project

Specifications, implementation, documentation, testing, distribution.

Several iterations

6 (optionally up to 9) steps, for 3 (resp. up to 6) months.

Algorithmically challenging

Applied use of well known data structures and algorithms.

Team Management

Project conducted in group of four students.

Goals

Aim

Compiler construction as a by-product

Complete Project

Specifications, implementation, documentation, testing, distribution.

Several iterations

6 (optionally up to 9) steps, for 3 (resp. up to 6) months.

Algorithmically challenging

Applied use of well known data structures and algorithms.

Team Management

Project conducted in group of four students.

Goals

Aim

Compiler construction as a by-product

Complete Project

Specifications, implementation, documentation, testing, distribution.

Several iterations

6 (optionally up to 9) steps, for 3 (resp. up to 6) months.

Algorithmically challenging

Applied use of well known data structures and algorithms.

Team Management

Project conducted in group of four students.

Goals

Aim

Compiler construction as a by-product

Complete Project

Specifications, implementation, documentation, testing, distribution.

Several iterations

6 (optionally up to 9) steps, for 3 (resp. up to 6) months.

Algorithmically challenging

Applied use of well known data structures and algorithms.

Team Management

Project conducted in group of four students.

Goals (cont.)

C++

Expressive power; uses both low- and high-level constructs; industry standard.

Object-Oriented (OO) Design and Design Pattern (DP)

Practice common OO idioms, apply DPs.

Development Tools

Autotools, Doxygen, Flex, Bison, GDB, Valgrind, Git, etc.

Understanding Computers

Compiler and languages are tightly related to computer architecture.

English

Everything is to be written in English (code, documentation, tests).

Goals (cont.)

C++

Expressive power; uses both low- and high-level constructs; industry standard.

Object-Oriented (OO) Design and Design Pattern (DP)

Practice common OO idioms, apply DPs.

Development Tools

Autotools, Doxygen, Flex, Bison, GDB, Valgrind, Git, etc.

Understanding Computers

Compiler and languages are tightly related to computer architecture.

English

Everything is to be written in English (code, documentation, tests).

Goals (cont.)

C++

Expressive power; uses both low- and high-level constructs; industry standard.

Object-Oriented (OO) Design and Design Pattern (DP)

Practice common OO idioms, apply DPs.

Development Tools

Autotools, Doxygen, Flex, Bison, GDB, Valgrind, Git, etc.

Understanding Computers

Compiler and languages are tightly related to computer architecture.

English

Everything is to be written in English (code, documentation, tests).

Goals (cont.)

C++

Expressive power; uses both low- and high-level constructs; industry standard.

Object-Oriented (OO) Design and Design Pattern (DP)

Practice common OO idioms, apply DPs.

Development Tools

Autotools, Doxygen, Flex, Bison, GDB, Valgrind, Git, etc.

Understanding Computers

Compiler and languages are tightly related to computer architecture.

English

Everything is to be written in English (code, documentation, tests).

Goals (cont.)

C++

Expressive power; uses both low- and high-level constructs; industry standard.

Object-Oriented (OO) Design and Design Pattern (DP)

Practice common OO idioms, apply DPs.

Development Tools

Autotools, Doxygen, Flex, Bison, GDB, Valgrind, Git, etc.

Understanding Computers

Compiler and languages are tightly related to computer architecture.

English

Everything is to be written in English (code, documentation, tests).

Non Goal

Writing a Compiler Paradoxically!

Well, at least considered a *secondary* issue.

- Why?

The vast majority of (Computer Science) students are unlikely to ever design a compiler (let alone a VM).

- But... Students interested in compiler construction should be given the opportunity to work on challenging, optional assignments.
- But... Since 2002, graphics processors units have raised (GPU). A lot of work has to be done to target such GPU.

Writing a Compiler Paradoxically!

Well, at least considered a *secondary* issue.

- Why?

The vast majority of [Computer Science] students are unlikely to ever design a compiler (Levy, 2002).

- But... Students interested in compiler construction should be given the opportunity to work on challenging, optional assignments.
- But... Since 2002, graphics processors units have raised (GPU). A lot of work has to be done to target such GPU.

Writing a Compiler Paradoxically!

Well, at least considered a **secondary** issue.

- Why?

The vast majority of [Computer Science] students are unlikely to ever design a compiler [Debray, 2002].

- But... Students interested in compiler construction should be given the opportunity to work on challenging, optional assignments.
- But... Since 2002, graphics processors units have raised (GPU). A lot of work has to be done to target such GPU.

Writing a Compiler Paradoxically!

Well, at least considered a *secondary* issue.

- Why?

The vast majority of [Computer Science] students are unlikely to ever design a compiler [Debray, 2002].

- But... Students interested in compiler construction should be given the opportunity to work on challenging, optional assignments.
- But... Since 2002, graphics processors units have raised (GPU). A lot of work has to be done to target such GPU.

Writing a Compiler Paradoxically!

Well, at least considered a *secondary* issue.

- Why?

The vast majority of [Computer Science] students are unlikely to ever design a compiler [Debray, 2002].

- But... Students interested in compiler construction should be given the opportunity to work on challenging, optional assignments.
- But... Since 2002, graphics processors units have raised (GPU). A lot of work has to be done to target such GPU.

Writing a Compiler Paradoxically!

Well, at least considered a *secondary* issue.

- Why?

The vast majority of [Computer Science] students are unlikely to ever design a compiler [Debray, 2002].

- But... Students interested in compiler construction should be given the opportunity to work on challenging, optional assignments.
- But... Since 2002, graphics processors units have raised (GPU). A lot of work has to be done to target such GPU.

Writing a Compiler Paradoxically!

Well, at least considered a *secondary* issue.

- Why?

The vast majority of [Computer Science] students are unlikely to ever design a compiler [Debray, 2002].

- But... Students interested in compiler construction should be given the opportunity to work on challenging, optional assignments.
- But... Since 2002, graphics processors units have raised (GPU). A lot of work has to be done to target such GPU.

Writing a Compiler Paradoxically!

Well, at least considered a *secondary* issue.

- Why?

The vast majority of [Computer Science] students are unlikely to ever design a compiler [Debray, 2002].

- But... Students interested in compiler construction should be given the opportunity to work on challenging, optional assignments.
- But... Since 2002, graphics processors units have raised (GPU). A lot of work has to be done to target such GPU.

Writing a Compiler Paradoxically!

Well, at least considered a *secondary* issue.

- Why?

The vast majority of [Computer Science] students are unlikely to ever design a compiler [Debray, 2002].

- But... Students interested in compiler construction should be given the opportunity to work on challenging, optional assignments.
- But... Since 2002, graphics processors units have raised (GPU). A lot of work has to be done to target such GPU.

Introduction to the Tiger Project

and Tigrou project

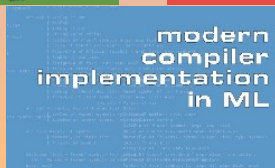
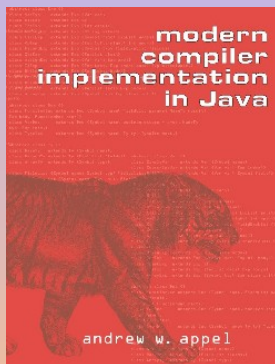
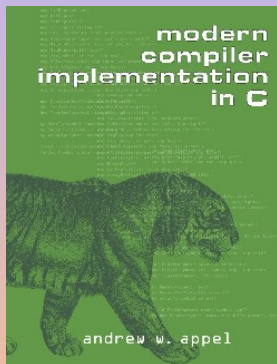
- 1 Overview of the Tiger Project
- 2 Practical information
- 3 Rules of the Game
- 4 Vocabulary and Structure of a compiler
- 5 The Tiger Compiler (and Tigrou's architecture)
- 6 The Tiger Language

Overview of the Tiger Project

- 1 Overview of the Tiger Project
- 2 Practical information
- 3 Rules of the Game
- 4 Vocabulary and Structure of a compiler
- 5 The Tiger Compiler (and Tigrou's architecture)
- 6 The Tiger Language

A Core-Curriculum Compiler-Construction Project

Based on Andrew Appel's Tiger language and *Modern Compiler Implementation* books [Appel, 1998].



A Core-Curriculum Compiler-Construction Project (cont.)

...and largely adapted [Demaille, 2005].

- Compiler (to be) written in C++.
- Initial Tiger language definition (a Pascal-descendant language, dressed in a clean ML-like syntax).
- Augmented with `import` statements, adjustable prelude, OO constructs, etc.
- Better defined (no implementation-defined behavior left).
- More compiler modules and features than in the initial design.
- In particular more *tools* to both help students develop and improve their compiler and make the maintenance easier to teachers and assistants.

A Core-Curriculum Compiler-Construction Project (cont.)

...and largely adapted [Demaille, 2005].

- **Compiler** (to be) written in C++.
- Initial Tiger language definition (a Pascal-descendant language, dressed in a clean ML-like syntax).
- Augmented with `import` statements, adjustable prelude, OO constructs, etc.
- Better defined (no implementation-defined behavior left).
- More compiler modules and features than in the initial design.
- In particular more *tools* to both help students develop and improve their compiler and make the maintenance easier to teachers and assistants.

A Core-Curriculum Compiler-Construction Project (cont.)

...and largely adapted [Demaille, 2005].

- **Compiler** (to be) written in C++.
- **Initial Tiger language definition** (a Pascal-descendant language, dressed in a clean ML-like syntax).
- Augmented with `import` statements, adjustable prelude, OO constructs, etc.
- Better defined (no implementation-defined behavior left).
- More compiler modules and features than in the initial design.
- In particular more *tools* to both help students develop and improve their compiler and make the maintenance easier to teachers and assistants.

A Core-Curriculum Compiler-Construction Project (cont.)

...and largely adapted [Demaille, 2005].

- **Compiler** (to be) written in C++.
- **Initial Tiger language definition** (a Pascal-descendant language, dressed in a clean ML-like syntax).
- Augmented with `import` statements, adjustable prelude, OO constructs, etc.
- Better defined (no implementation-defined behavior left).
- More compiler modules and features than in the initial design.
- In particular more *tools* to both help students develop and improve their compiler and make the maintenance easier to teachers and assistants.

Project's *Modus Operandi*

- The compiler is designed as a long pipe composed of several modules.
- The project is divided in several steps, where students have to implement one (or two) module(s).
- Code with gaps.
- Work is evaluated by a program at each delivery.
- Students defend their work every two steps in front of a teaching assistant.
- Several optional assignments are given as extra modules.
- Motivated students can choose to proceed with the implementation of the back-end of the compiler.

Project's *Modus Operandi*

- The compiler is designed as a long pipe composed of several modules.
- The project is divided in several steps, where students have to implement one (or two) module(s).
- Code with gaps.
- Work is evaluated by a program at each delivery.
- Students defend their work every two steps in front of a teaching assistant.
- Several optional assignments are given as extra modules.
- Motivated students can choose to proceed with the implementation of the back-end of the compiler.

Project's *Modus Operandi*

- The compiler is designed as a long pipe composed of several modules.
- The project is divided in several steps, where students have to implement one (or two) module(s).
- Code with gaps.
- Work is evaluated by a program at each delivery.
- Students defend their work every two steps in front of a teaching assistant.
- Several optional assignments are given as extra modules.
- Motivated students can choose to proceed with the implementation of the back-end of the compiler.

Project's *Modus Operandi*

- The compiler is designed as a long pipe composed of several modules.
- The project is divided in several steps, where students have to implement one (or two) module(s).
- Code with gaps.
- Work is evaluated by a program at each delivery.
- Students defend their work every two steps in front of a teaching assistant.
- Several optional assignments are given as extra modules.
- Motivated students can choose to proceed with the implementation of the back-end of the compiler.

Figures

- 20 years of existence.
- 250/300 students per year (on average).
- Project done in groups of 4 students.
- 6 mandatory steps (compiler front-end).
- 4 optional steps (compiler back-end).
- Reference compiler: 25KLOC.
- Students are expected to write about 5500 lines (or about 7000 lines, with the optional assignments).
- 250+ pages of documentation (reference manual [Demaille and Levillain, 2007b] and project assignments [Demaille and Levillain, 2007a]).
- 3 papers in international conferences [Demaille, 2005, Demaille et al., 2008, Demaille et al., 2009].

History I

- 2000 Beginning of the Tiger Project: a front-end, a single teacher, no assistant.
- 2001 Have students learn and use the Autotools for project maintenance.
- 2002 Teaching Assistants involved in the project.
Interpreter for the Intermediate Representation (IR) language (HAVM).
- 2003 Addition of a MIPS back-end, partly from the work of motivated students.
Interpreter for the MIPS language (Nolimips).
The structures of the Abstract Syntax Tree (AST) and a visitor are generated from a description file.

History II

- 2005 A second teacher in the project maintenance and supervision.
First uses of some Boost libraries (Boost.Variant, Boost Graph Library (BGL), Smart Pointers).
First use of concrete-syntax program transformations (code generation) using TWEASTs
- 2007 Tiger becomes an Object-Oriented Language (OOL).
- 2009 C++ objects on the parser stack.
- 2011 Extension of Bison's grammar to handle named parameters.

History I

- 2012 Conversion of tc to C++11.
- 2015 Even more C++11/C++14, aiming at C++17.
 - OO is now optional.
 - Much simpler/faster build system.
 - Support from ARM backend.
- 2016 Support for LLVM intermediate language.
- 2017 Introduce Dockerfile.
 - Move on C++17
- 2017 Introduce Dockerfile.
- 2018 Refactoring python bindings for jupyter-notebooks
- 2019 Rework assignments
- 2020 Setup Tigrou's project

Practical information

- 1 Overview of the Tiger Project
- 2 Practical information**
- 3 Rules of the Game
- 4 Vocabulary and Structure of a compiler
- 5 The Tiger Compiler (and Tigrou's architecture)
- 6 The Tiger Language

Entry Points

- Home page of the project:
`http://tiger.lrde.epita.fr`
- Resources for the project:
`http://www.lrde.epita.fr/~tiger`
- `news.epita.fr` newsgroup:
`assistants.tiger`

- Tiger Compiler Project Assignments:
<http://assignments.lrde.epita.fr/>
- Tiger Compiler Reference Manual (TCRM):
<http://www.lrde.epita.fr/~tiger/tiger.html>
<http://www.lrde.epita.fr/~tiger/tiger.split>
<http://www.lrde.epita.fr/~tiger/tiger.pdf>
<http://www.lrde.epita.fr/~tiger/tiger.info>
<http://www.lrde.epita.fr/~tiger/tiger.txt>

Slides (like these ones):

<http://www.lrde.epita.fr/~tiger/lecture-notes/slides/>

Handouts (if you need to print lectures, please use these and save a tree):

1-up <http://www.lrde.epita.fr/~tiger/lecture-notes/handouts/>

4-up <http://www.lrde.epita.fr/~tiger/lecture-notes/handouts-4/>

Subdirectories:

[ccmp/](#) CCMP lecture notes

[tc/](#) TC- n presentations, given by the Assistants

[tyla/](#) TYLA lecture notes

Other Useful Resources

- Doxygen Documentation:
<http://www.lrde.epita.fr/~tiger/tc-doc/>
- Past exams [outdated since 2016]
<http://www.lrde.epita.fr/~tiger/exams/>
- Emacs/Vim modes
<http://www.lrde.epita.fr/~tiger/tc/tiger.el>
<http://www.lrde.epita.fr/~tiger/tc/tiger.vim>
- Docker for a fully working environnement
<http://www.lrde.epita.fr/~tiger/tc/docker-sid>

Rules of the Game

- 1 Overview of the Tiger Project
- 2 Practical information
- 3 Rules of the Game**
- 4 Vocabulary and Structure of a compiler
- 5 The Tiger Compiler (and Tigrou's architecture)
- 6 The Tiger Language

Nul n'est censé ignorer la loi.

`http://www.assignements.lrde.epita.fr`

- ❶ **No copy between groups.**
- ❷ Tests are part of the project
(test cases and frameworks should not be exchanged).
- ❸ Fixing mistakes earlier is better (and less expensive).
- ❹ Work between groups is encouraged!
As long as they don't cheat.

Nul n'est censé ignorer la loi.

`http://www.assignements.lrde.epita.fr`

- 1 **No copy between groups.**
- 2 **Tests are part of the project**
(test cases and frameworks should not be exchanged).
- 3 Fixing mistakes earlier is better (and less expensive).
- 4 Work between groups is encouraged!
As long as they don't cheat.

Nul n'est censé ignorer la loi.

`http://www.assignements.lrde.epita.fr`

- ❶ **No copy between groups.**
- ❷ **Tests are part of the project**
(test cases and frameworks should not be exchanged).
- ❸ **Fixing mistakes earlier is better (and less expensive).**
- ❹ Work between groups is encouraged!
As long as they don't cheat.

Nul n'est censé ignorer la loi.

`http://www.assignements.lrde.epita.fr`

- 1 **No copy between groups.**
- 2 **Tests are part of the project**
(test cases and frameworks should not be exchanged).
- 3 **Fixing mistakes earlier is better (and less expensive).**
- 4 **Work between groups is encouraged!**
As long as they don't cheat.

Vocabulary and Structure of a compiler

- 1 Overview of the Tiger Project
- 2 Practical information
- 3 Rules of the Game
- 4 Vocabulary and Structure of a compiler
- 5 The Tiger Compiler (and Tigrou's architecture)
- 6 The Tiger Language

- A **compiler** is a program that converts a source language into a target machine language.
- A **cross-compiler** is a program (running on a machine A) that converts a source language into a target machine language B (different from A).
- A **transpiler** is a program that converts a source language into a target language (same level of abstraction).

- **Pre-processor** remove all *define*, *include* directives. Produces a *pure* code.
- **Linker** combines one or more object files and possible some library code into either some executable, or some library
- **Loader** A loader reads the executable code into memory, does some address translation and tries to run the program resulting in a running program.
- Linker and loader performs Basically the loader does the program loading; the linker does the symbol resolution; and either of them can do the relocation.
- **Static/Dynamic** Compile time / runtime

The C Compilation Model

Which (coarse-grained) steps can we find in gcc?

The C Compilation Model

Which (coarse-grained) steps can we find in gcc?

- cpp (preprocessor)

The C Compilation Model

Which (coarse-grained) steps can we find in gcc?

- cpp (preprocessor)
- cc1 (actual C compiler)

The C Compilation Model

Which (coarse-grained) steps can we find in gcc?

- cpp (preprocessor)
- cc1 (actual C compiler)
- as (assembler)

The C Compilation Model

Which (coarse-grained) steps can we find in gcc?

- cpp (preprocessor)
- cc1 (actual C compiler)
- as (assembler)
- ld (linker)

The C Compilation Model

Which (coarse-grained) steps can we find in gcc?

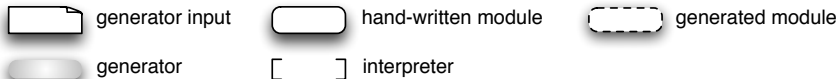
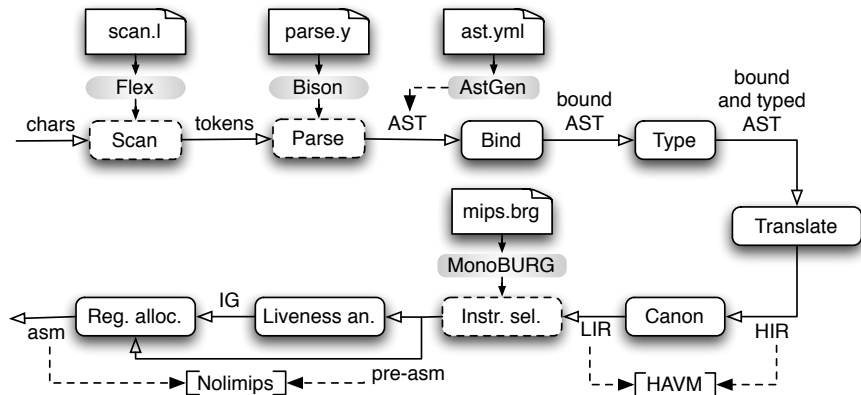
- cpp (preprocessor)
- cc1 (actual C compiler)
- as (assembler)
- ld (linker)

→ A *pipe*.

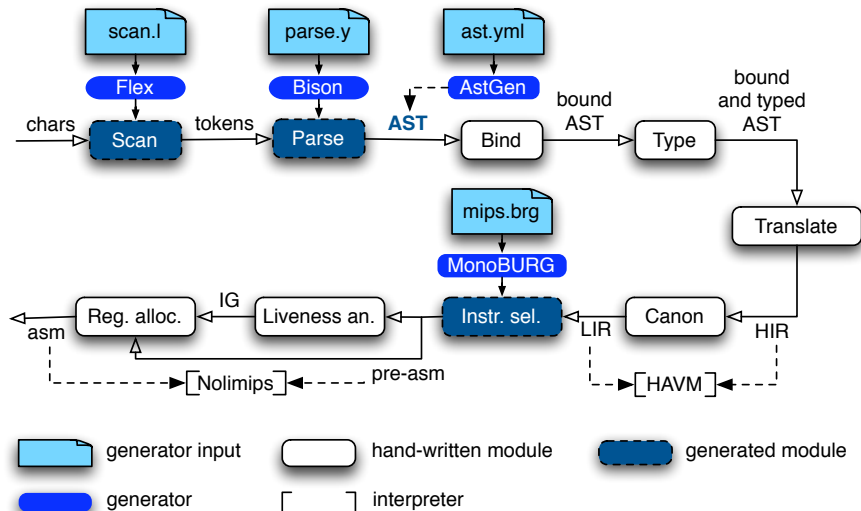
The Tiger Compiler (and Tigrou's architecture)

- 1 Overview of the Tiger Project
- 2 Practical information
- 3 Rules of the Game
- 4 Vocabulary and Structure of a compiler
- 5 The Tiger Compiler (and Tigrou's architecture)**
- 6 The Tiger Language

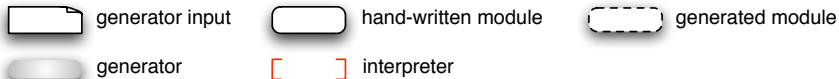
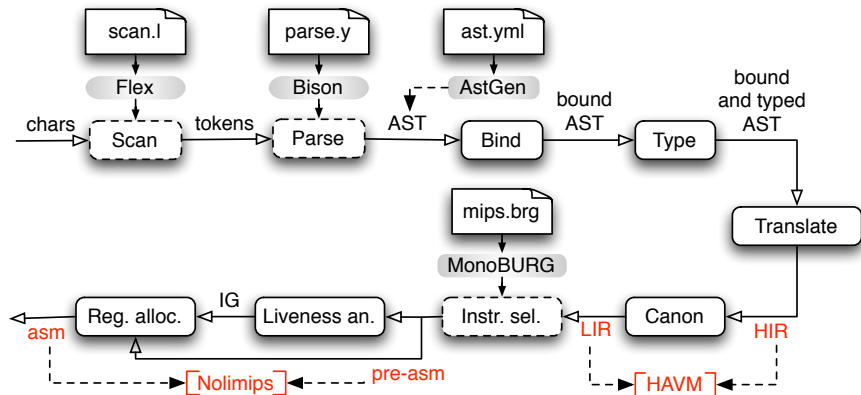
tc: A Compiler as A Long Pipe



tc: A Compiler as A Long Pipe



tc: A Compiler as A Long Pipe



Other Compiling Strategies

- Intermediate language-based strategy: SmartEiffel, GHC
- Bytecode strategy: Java bytecode (JVM), CIL (.NET)
- Hybrid approaches: GCJ (Java bytecode or native code)
- Retargetable optimizing back ends: MLRISC, VPO (Very Portable Optimizer), and somehow C-- (Quick C--).
- Modular systems: LLVM (compiler as a library, centered on a typed IR). Contains the LLVM core libraries, Clang, LLDB, etc. Also:
 - VMKit: a substrate for virtual machines (JVM, etc.).
 - Emscripten: an LLVM-to-JavaScript compiler. Enables C/C++ to JS compilation.

Intermediate Representations (IR) are fundamental.

The Tiger Language

- 1 Overview of the Tiger Project
- 2 Practical information
- 3 Rules of the Game
- 4 Vocabulary and Structure of a compiler
- 5 The Tiger Compiler (and Tigrou's architecture)
- 6 The Tiger Language

Two flavors

Appel's Defined in *Modern Compiler Implementation* books
(see the Appendix).

Ours Defined in the Tiger Compiler Reference Manual (TCRM).

- Features many extensions: export keyword, overloading, OOP, ...
- Implemented by LRDE's reference compiler.
- This is the target language for your project.

Two flavors

Appel's Defined in *Modern Compiler Implementation* books
(see the Appendix).

Ours Defined in the Tiger Compiler Reference Manual (TCRM).

- Features many extensions: `import` keyword, overloading, OOP, ...
- Implemented by LRDE's reference compiler.
- This is the target language for your project.

Two flavors

Appel's Defined in *Modern Compiler Implementation* books
(see the Appendix).

Ours Defined in the Tiger Compiler Reference Manual (TCRM).

- Features many extensions: **import** keyword, overloading, OOP, ...
- Implemented by LRDE's reference compiler.
- This is the target language for your project.

Two flavors

Appel's Defined in *Modern Compiler Implementation* books
(see the Appendix).

Ours Defined in the Tiger Compiler Reference Manual (TCRM).

- Features many extensions: `import` keyword, overloading, OOP, ...
- Implemented by LRDE's reference compiler.
- This is the target language for your project.

Two flavors

Appel's Defined in *Modern Compiler Implementation* books
(see the Appendix).

Ours Defined in the Tiger Compiler Reference Manual (TCRM).

- Features many extensions: `import` keyword, overloading, OOP, ...
- Implemented by LRDE's reference compiler.
- **This is the target language for your project.**

- Toy language (not industry-proof)...
...but still effective.
- Imperative language, descendant of Algol.
- Functional flavour.
 - `var x = 1; var y = 2; var z`
 - `function incr(x : int) : int = x + 1`
- Simple and sound grammar.
- Well defined semantics.

- Toy language (not industry-proof)...
...but still effective.
- Imperative language, descendant of Algol.
- Functional flavour.
 - `if x = 12 then 2 else 3`
 - `function incr(x : int) : int = x + 1`
- Simple and sound grammar.
- Well defined semantics.

- Toy language (not industry-proof)...
...but still effective.
- Imperative language, descendant of Algol.
- Functional flavour.
 - `a := if 1 then 2 else 3`
 - `function incr(x : int) : int = x + 1`
- Simple and sound grammar.
- Well defined semantics.

- Toy language (not industry-proof)...
...but still effective.
- Imperative language, descendant of Algol.
- Functional flavour.

- `a := if 1 then 2 else 3`
 - `function incr(x : int) : int = x + 1`

- Simple and sound grammar.
- Well defined semantics.

- Toy language (not industry-proof)...
...but still effective.
- Imperative language, descendant of Algol.
- Functional flavour.
 - `a := if 1 then 2 else 3`
 - `function incr(x : int) : int = x + 1`
- Simple and sound grammar.
- Well defined semantics.

- Toy language (not industry-proof)...
...but still effective.
- Imperative language, descendant of Algol.
- Functional flavour.
 - `a := if 1 then 2 else 3`
 - `function incr(x : int) : int = x + 1`
- Simple and sound grammar.
- Well defined semantics.

- Toy language (not industry-proof)...
...but still effective.
- Imperative language, descendant of Algol.
- Functional flavour.
 - `a := if 1 then 2 else 3`
 - `function incr(x : int) : int = x + 1`
- Simple and sound grammar.
- Well defined semantics.

- Toy language (not industry-proof)...
...but still effective.
- Imperative language, descendant of Algol.
- Functional flavour.
 - `a := if 1 then 2 else 3`
 - `function incr(x : int) : int = x + 1`
- Simple and sound grammar.
- Well defined semantics.

Your first Tiger Program

```
print("Hello World!\n")
```

Your second Tiger program

```
let
  function hello(name : string) =
    print(concat("Hello ", name))
in
  hello("You");
  print("\n")
end
```

The classic Factorial function

```
let
  /* Compute n! */
  function fact(n : int) : int =
    if n = 0
      then 1
      else n * fact(n - 1)
in
  print_int(fact(10));
  print("\n")
end
```

Bibliography I



Appel, A. W. (1998).

Modern Compiler Implementation in C, Java, ML.

Cambridge University Press.



Debray, S. (2002).

Making compiler design relevant for students who will (most likely) never design a compiler.

In Proceedings of the 33rd SIGCSE technical symposium on Computer science education, pages 341–345. ACM Press.



Demaille, A. (2005).

Making compiler construction projects relevant to core curriculums.

In Proceedings of the Tenth Annual Conference on Innovation and Technology in Computer Science Education (ITICSE'05), pages 266–270, Universidade Nova de Lisboa, Monte da Pacarita, Portugal.



Demaille, A. and Levillain, R. (2007a).

The Tiger Compiler Project Assignment.

EPITA Research and Development Laboratory (LRDE), 14-16 rue
Voltaire, FR-94270 Le Kremlin-Bicêtre, France.

<http://www.lrde.epita.fr/~akim/ccmp/assignments.pdf>.



Demaille, A. and Levillain, R. (2007b).

The Tiger Compiler Reference Manual.

EPITA Research and Development Laboratory (LRDE), 14-16 rue
Voltaire, FR-94270 Le Kremlin-Bicêtre, France.

<http://www.lrde.epita.fr/~akim/ccmp/tiger.pdf>.



Demaille, A., Levillain, R., and Perrot, B. (2008).

A set of tools to teach compiler construction.

In *Proceedings of the Thirteenth Annual Conference on Innovation and Technology in Computer Science Education (ITICSE'08)*, pages 68–72, Universidad Politécnica de Madrid, Spain.



Demaille, A., Levillain, R., and Sigoure, B. (2009).

TWEAST: A simple and effective technique to implement concrete-syntax AST rewriting using partial parsing.

In *Proceedings of the 24th Annual ACM Symposium on Applied Computing (SAC'09)*, pages 1924–1929, Waikiki Beach, Honolulu, Hawaii, USA.

The Scanner and The Parser

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

EPITA — École Pour l'Informatique et les Techniques Avancées

February 2, 2020

The Scanner and The Parser

- 1 Flex & Bison: Recalls
- 2 Semantic Values
- 3 Locations
- 4 Improving the Scanner/Parser
- 5 Symbols

Flex & Bison: Recalls

- 1 Flex & Bison: Recalls
- 2 Semantic Values
- 3 Locations
- 4 Improving the Scanner/Parser
- 5 Symbols

Flex:

- Lexical analyser
- Generates **scanners**
- Description in the form of regular expressions

Structure

```
%{  
    [definitions]  
%}  
%%  
[rules]  
%%  
[subprograms]
```

- Work on regular expressions **ONLY**
- Define regexps
 - Letter [a-zA-Z]
 - Number [0-9]
 - ...
- **yytext** the recognized text
- **yylen** the size of the recognized text
- **yylex** starts the scanning
- **yywrap** called when the end of the text to analyze is encountered.
Can be refined if needed.
- For each of matched regexps one can return an identifier (a token)
- *Bison will analyze this stream of tokens...*
 - Details later in this lecture for coupling flex and bison

Flex example – wc linux command

```
%{
#include <stdio.h>
static int chars_ = 0, lines_ = 0, words_ = 0;
}%

%%
\n      { ++chars_; ++lines_; }
[^\t\n]+ { chars_ += yyleng; ++words_; }
.       { ++chars_; }
%%
int yywrap () {
    printf ("%7d %7d %7d\n", lines_, words_, chars_);
    return 1;
}
int main(){  yylex(); return 1; }
```

Bison¹:

- Syntactic analyser
- Generates **parser**
- hand-by-hand with flex: read token to analyse the input stream

Structure

```
[definitions]
%%
[rules]
%%
[%%
  subprograms
]
```

¹One should note that for the project we use a patched version of bison that supports variants www.lrde.epita.fr/~tiger/download/bison-3.2.1.52-cd4f7.tar.gz 

LALR vs. GLR

LALR-1

- Default for bison
- Default behavior when a conflict occurs:
 - reduce/reduce: reduce to the first rule in conflict
 - shift/reduce: performs the shift
- During a shift/reduce conflict the parser may *miles away from the ball*

GLR

- 1 During a conflict the parser walks the two branches hoping that one of the two will win.
- 2 Maintains multiple parse stacks
- 3 Allows ambiguous grammars (when required by the language)

Ambiguous grammar

- 1 **Ambiguous grammar:** the parser cannot choose
- 2 **one branch succeeds:** the parser choose this one
- 3 **Syntax Error:** easy case, report error!

Ambiguous grammar: example 1

```
%%  
exp:  
    "if" exp "then" exp  
    | "if" exp "then" exp "else" exp  
    | "exp"  
    ;  
%%
```

Problem: Dangling Else

"else" should attach to which "if"? Inner one or outer one?

if "exp" then if "exp" then "exp" else "exp"

Ambiguous grammar: example 1

```
%%  
exp:  
    "if" exp "then" exp  
    | "if" exp "then" exp "else" exp  
    | "exp"  
    ;  
%%
```

Problem: Dangling Else

"else" should rattach to which "if"? Inner one or outer one?

if "exp" then if "exp" then "exp" else "exp"

Ambiguous grammar: example 1 solution

```
%expect 0
%right "else" "then"
%%
exp:
    "if" exp "then" exp
  | "if" exp "then" exp "else" exp
  | "exp"
  ;
%%
```

- %right: choose shift
- %left: choose reduce
- %expect: the number of expected conflicts

Another solution would be to add "fi".

Ambiguous grammar: example 2

```
%%  
exp:  
    exp "?" exp ":" exp  
    | "exp"  
    ;  
%%
```

Problem: Dangling ":"

":" should attach to which "?" ? Inner one or outer one?

"exp" ? "exp" ? "exp" : "exp"

Ambiguous grammar: example 2

```
%%  
exp:  
    exp "?" exp ":" exp  
    | "exp"  
    ;  
%%
```

Problem: Dangling ":"

":" should attach to which "?" ? Inner one or outer one?

"exp" ? "exp" ? "exp" : "exp"

Ambiguous grammar: example 2 solution

```
%expect 0
%right ":" "?"
%%
exp:
    exp "?" exp ":" exp
    | exp
    ;
%%
```

Ambiguous grammar: example 3

```
%%  
exp:  
    typeid "[" exp "]" "of" exp  
    | lvalue  
    ;  
  
lvalue:  
    "id"  
    | lvalue "[" exp "]"  
    ;  
  
typeid:  
    "id"  
    ;  
%%
```

Ambiguous grammar: example 3 – problems

Problems

- typeid must be removed and "id" must be propagated

`"id" [ "id" [ "id" [ "id" ] ] ]`

- lvalues can be nested and the decision is taken on the “of” which is too late! There must be no question between "typeid" and "["

Ambiguous grammar: example 3 solution

```
%%  
exp:  
    "id" "[" exp "]" "of" exp  
    | lvalue  
    ;  
  
lvalue:  
    "id"  
    | lvalue_b  
    ;  
  
lvalue_b:  
    "id" of "[" exp "]"  
    | lvalue_b "[" exp "]"  
    ;  
%%
```

Semantic Values

1 Flex & Bison: Recalls

2 Semantic Values

- Coupling Parser and Scanner
- Parser
- Scanner

3 Locations

4 Improving the Scanner/Parser

5 Symbols

Coupling Parser and Scanner

1 Flex & Bison: Recalls

2 Semantic Values

- Coupling Parser and Scanner
- Parser
- Scanner

3 Locations

4 Improving the Scanner/Parser

5 Symbols

Objectives

How to produce a stream of tokens in the scanner that will be analyzed by the parser?

Steps:

- 1 define token in the *parser.yy* using *%token TOKENNAME*
- 2 bison will produce an header file **that should be included into your scanner**
 - Your scanner can now see declared tokens
 - When the scanner match a regexp return to associated token in the flex's rule

Note: *%token < XXX > TOKENNAME* associates a token to a value (here XXX).

Demo Time!

Variants (or how to move to C++)

The parser maintains a stack of types.

- In C, no problem use a **union**
- In C++, ...???

⇒ solution: **variants**

Variants

Variants are type safe unions:

- allocated directly within the object representation of the variant
- call destructors
- bison implements such a variants where the stack maintains the type (to call the correct destructor).

Variants (or how to move to C++)

The parser maintains a stack of types.

- In C, no problem use a **union**
- In C++, ...???

⇒ solution: **variants**

Variants

Variants are type safe unions:

- allocated directly within the object representation of the variant
- call destructors
- bison implements such a variants where the stack maintains the type (to call the correct destuctor).

- 1 Flex & Bison: Recalls
- 2 Semantic Values
 - Coupling Parser and Scanner
 - Parser
 - Scanner
- 3 Locations
- 4 Improving the Scanner/Parser
- 5 Symbols

Reading tokens in the parser

```
// Allow storing object values.
%define api.value.type variant
// Generate functions to build tokens.
%define api.token.constructor
// Prefix all the tokens with TOK_ to avoid colisions.
%define api.token.prefix {TOK_}

%token <misc::symbol>    ID      "identifier"
%token <int>             INT      "integer"
%token <std::string>     STRING  "string"

%printer { yyo << $$; } "identifier" "integer" "string"
%%
// ...
exp:
    INT      { $$ = new IntExp($1); }
| STRING    { $$ = new StringExp($1); }
//...
```

- 1 Flex & Bison: Recalls
- 2 Semantic Values
 - Coupling Parser and Scanner
 - Parser
 - **Scanner**
- 3 Locations
- 4 Improving the Scanner/Parser
- 5 Symbols

Generating tokens from the scanner

```
id      [a-zA-Z][a-zA-Z_0-9]*
int     [0-9]+
string  "\\"([^\\"|\\\.)*\\""
```

%%

```
{id}      return parser::make_ID(yytext);
{int}     return parser::make_INT(atoi(yytext));
{string}  return parser::make_STRING(std::string(yytext + 1,
                                                    yyleng - 2));
```

or even (C++ 11)

```
{string}  return parser::make_STRING({yytext+1, yyleng-2});
```

Locations

- 1 Flex & Bison: Recalls
- 2 Semantic Values
- 3 Locations**
 - Location tracking in the Scanner
 - Location tracking in the Parser
- 4 Improving the Scanner/Parser
- 5 Symbols

Location tracking in the Scanner

- 1 Flex & Bison: Recalls
- 2 Semantic Values
- 3 Locations**
 - Location tracking in the Scanner
 - Location tracking in the Parser
- 4 Improving the Scanner/Parser
- 5 Symbols

Location tracking in Flex

What

`loc` the current location

How

`%initial-action`

run at the beginning of `yyparse`

`%USER_ACTION`

once per scanner match

`%{ ... %}`

(after the first `%`) pasted into `yylex`.

When at its top when first in the rule section:

- local variables
- code run once per `yylex` invocation

Location tracking in Flex

What

`loc` the current location

How

`%initial-action`

run at the beginning of `yyparse`.

`%initial-action {`

once per scanner match

`%{ ... %}`

(after the first `%`) pasted into `yylex`.

When at its top when first in the rule section:

- local variables
- code run once per `yylex` invocation

Location tracking in Flex

What

`loc` the current location

How

`%initial-action`

run at the beginning of `yyparse`.

`YY_USER_ACTION`

once per scanner match

`%{ ... %}`

(after the first `%`) pasted into `yylex`.

When at its top when first in the rule section:

- local variables
- code run once per `yylex` invocation

Location tracking in Flex

What

`loc` the current location

How

`%initial-action`

run at the beginning of `yyparse`.

`YY_USER_ACTION`

once per scanner match

`%{ ... %}`

(after the first `%%`) pasted into `yylex`.

When at its top when first in the rule section:

- local variables
- code run once per `yylex` invocation

Location tracking in Flex

What

`loc` the current location

How

`%initial-action`

run at the beginning of `yyparse`.

`YY_USER_ACTION`

once per scanner match

`%{ ... %}`

(after the first `%`) pasted into `yylex`.

When at its top when first in the rule section:

- local variables
- code run once per `yylex` invocation

Location tracking in Flex

What

`loc` the current location

How

`%initial-action`

run at the beginning of `yyparse`.

`YY_USER_ACTION`

once per scanner match

`%{ ... %}`

(after the first `%%`) pasted into `yylex`.

When at its top when first in the rule section:

- local variables
- code run once per `yylex` invocation

Location tracking in Flex

What

`loc` the current location

How

`%initial-action`

run at the beginning of `yyparse`.

`YY_USER_ACTION`

once per scanner match

`%{ ... %}`

(after the first `%%`) pasted into `yylex`.

When at its top when first in the rule section:

- local variables
- code run once per `yylex` invocation

Location tracking in Flex

What

`loc` the current location

How

`%initial-action`

run at the beginning of `yyparse`.

`YY_USER_ACTION`

once per scanner match

`%{ ... %}`

(after the first `%`) pasted into `yylex`.

When at its top when first in the rule section:

- local variables
- code run once per `yylex` invocation

Location tracking in Flex

```
%{  
    /* At each match, adjust the last column. */  
# define YY_USER_ACTION  loc.columns(yyleng);  
%}  
/* ... */  
%%  
%{  
    /* At each call, bring the tail to the head.  */  
    loc.step();  
%}  
  
    /* Locations of blanks are ignored. */  
[ \t]+  loc.step();  
  
    /* Newlines change the current line number,  
       but are ignored too. */  
\n+    loc.line(yyleng); loc.step();
```

Location tracking in Flex

```
{id}      return parser::make_ID(yytext, loc);  
{int}     return parser::make_INT(atoi(yytext), loc);  
{string}  return parser::make_STRING({yytext+1, yyleng-2}, loc);
```

Location tracking in the Parser

- 1 Flex & Bison: Recalls
- 2 Semantic Values
- 3 Locations**
 - Location tracking in the Scanner
 - **Location tracking in the Parser**
- 4 Improving the Scanner/Parser
- 5 Symbols

Using the Location in the Parser

```
%define filename_type {const std::string}
%locations

%%

lvalue.big:
    ID "[" exp "]"
    { $$ = new SubscriptVar
        (@$, new SimpleVar(@1, $1), $3); }
| lvalue.big "[" exp "]"
    { $$ = new SubscriptVar(@$, $1, $3); }
;
```

Error Messages

```
%error-verbose
%%
// ...
%%
void
yy::parser::error(const location_type& l, const std::string& m)
{
    tp.error_ << misc::Error::parse
               << l << ": " << m << std::endl;
}
```

Improving the Scanner/Parser

- 1 Flex & Bison: Recalls
- 2 Semantic Values
- 3 Locations
- 4 Improving the Scanner/Parser
 - Error Recovery
 - Pure Parser
 - Two Grammars in One
 - Reentrancy
- 5 Symbols

Error Recovery

- 1 Flex & Bison: Recalls
- 2 Semantic Values
- 3 Locations
- 4 Improving the Scanner/Parser**
 - Error Recovery
 - Pure Parser
 - Two Grammars in One
 - Reentrancy
- 5 Symbols

- The error token in Yacc/Bison:
 - ① dig in the stack to find a nice place
 - ② throw away unpleasant lookaheads
 - ③ reduce as usual
- “Guard” it, put bounds around
- May introduce new conflicts.
- Do as if there were no error: generate dummy values
- Maybe introduce an `Error` class to prevent cascades of errors.

- The error token in Yacc/Bison:
 - 1 dig in the stack to find a nice place
 - 2 throw away unpleasant lookaheads
 - 3 reduce as usual
- “Guard” it, put bounds around
- May introduce new conflicts.
- Do as if there were no error: generate dummy values
- Maybe introduce an `Error` class to prevent cascades of errors.

- The error token in Yacc/Bison:
 - 1 dig in the stack to find a nice place
 - 2 throw away unpleasant lookaheads
 - 3 reduce as usual
- “Guard” it, put bounds around
- May introduce new conflicts.
- Do as if there were no error: generate dummy values
- Maybe introduce an `Error` class to prevent cascades of errors.

- The error token in Yacc/Bison:
 - 1 dig in the stack to find a nice place
 - 2 throw away unpleasant lookaheads
 - 3 reduce as usual
- “Guard” it, put bounds around
- May introduce new conflicts.
- Do as if there were no error: generate dummy values
- Maybe introduce an `Error` class to prevent cascades of errors.

- The error token in Yacc/Bison:
 - 1 dig in the stack to find a nice place
 - 2 throw away unpleasant lookaheads
 - 3 reduce as usual
- “Guard” it, put bounds around
 - May introduce new conflicts.
 - Do as if there were no error: generate dummy values
 - Maybe introduce an Error class to prevent cascades of errors.

- The error token in Yacc/Bison:
 - 1 dig in the stack to find a nice place
 - 2 throw away unpleasant lookaheads
 - 3 reduce as usual
- “Guard” it, put bounds around
- May introduce new conflicts.
- Do as if there were no error: generate dummy values
- Maybe introduce an `Error` class to prevent cascades of errors.

- The error token in Yacc/Bison:
 - 1 dig in the stack to find a nice place
 - 2 throw away unpleasant lookaheads
 - 3 reduce as usual
- “Guard” it, put bounds around
- May introduce new conflicts.
- Do as if there were no error: generate dummy values
- Maybe introduce an `Error` class to prevent cascades of errors.

- The error token in Yacc/Bison:
 - 1 dig in the stack to find a nice place
 - 2 throw away unpleasant lookaheads
 - 3 reduce as usual
- “Guard” it, put bounds around
- May introduce new conflicts.
- Do as if there were no error: generate dummy values
- Maybe introduce an Error class to prevent cascades of errors.

```
parse/parsetiger.yy
```

```
// Reclaim the memory.
%destructor { delete $$; } exp
%%
exp:
    "nil"          { $$ = new NilExp(@$); }
| "(" exps ")"    { $$ = new SeqExp(@$, $2); }
| "(" error ")"   { $$ = new SeqExp(@$, new exps_t); }
// ...
```

- 1 Flex & Bison: Recalls
- 2 Semantic Values
- 3 Locations
- 4 Improving the Scanner/Parser**
 - Error Recovery
 - **Pure Parser**
 - Two Grammars in One
 - Reentrancy
- 5 Symbols

The Parsing Driver

- Information exchanged with the parser/scanner
 - Input data
library path, debugging flags, etc.
 - Output data
The ast, the error messages/status
 - Data maintained during the parsing
Open files
- Coordination
 - Initialize/open the scanner
 - Parse
 - Close the scanner
- Introduce a parsing driver.

The Parsing Driver

- Information exchanged with the parser/scanner
 - Input data
library path, debugging flags, etc.
 - Output data
The ast, the error messages/status
 - Data maintained during the parsing
Open files
- Coordination
 - Initialize/open the scanner
 - Parse
 - Close the scanner
- Introduce a parsing driver.

The Parsing Driver

- Information exchanged with the parser/scanner
 - Input data
library path, debugging flags, etc.
 - Output data
The ast, the error messages/status
 - Data maintained during the parsing
Open files
- Coordination
 - Initialize/open the scanner
 - Parse
 - Close the scanner
- Introduce a parsing driver.

The Parsing Driver

- Information exchanged with the parser/scanner
 - Input data
library path, debugging flags, etc.
 - Output data
The ast, the error messages/status
 - Data maintained during the parsing
Open files
- Coordination
 - Initialize/open the scanner
 - Parse
 - Close the scanner
- Introduce a parsing driver.

The Parsing Driver

- Information exchanged with the parser/scanner
 - Input data
library path, debugging flags, etc.
 - Output data
The ast, the error messages/status
 - Data maintained during the parsing
Open files
- Coordination
 - Initialize/open the scanner
 - Parse
 - Close the scanner
- Introduce a parsing driver.

The Parsing Driver

- Information exchanged with the parser/scanner
 - Input data
library path, debugging flags, etc.
 - Output data
The ast, the error messages/status
 - Data maintained during the parsing
Open files
- Coordination
 - Initialize/open the scanner
 - Parse
 - Close the scanner
- Introduce a parsing driver.

The Parsing Driver

- Information exchanged with the parser/scanner
 - Input data
library path, debugging flags, etc.
 - Output data
The ast, the error messages/status
 - Data maintained during the parsing
Open files
- Coordination
 - Initialize/open the scanner
 - Parse
 - Close the scanner
- Introduce a parsing driver.

The Parsing Driver

- Information exchanged with the parser/scanner
 - Input data
library path, debugging flags, etc.
 - Output data
The ast, the error messages/status
 - Data maintained during the parsing
Open files
- Coordination
 - Initialize/open the scanner
 - Parse
 - Close the scanner
- Introduce a parsing driver.

The Parsing Driver

- Information exchanged with the parser/scanner
 - Input data
library path, debugging flags, etc.
 - Output data
The ast, the error messages/status
 - Data maintained during the parsing
Open files
- Coordination
 - Initialize/open the scanner
 - Parse
 - Close the scanner
- Introduce a **parsing driver**.

The Parsing Driver (parse/tiger-parser.hh)

```
class TigerParser
{
public:
    /// Parse a Tiger program, return its AST.
    ast::Exp* parse_program(...);
    /// Parse a Tiger prelude, return the list of decs.
    ast::decs_list_type* parse_import(...);

private:
    /// The result of the parse.
    ast_type ast_;
    /// Parsing errors handler.
    misc::error error_;
    /// The source to parse.
    input_type input_;
    /// The file library for imports.
    misc::file_library library_;
};
```


The Parsing Driver (parse/tiger-parser.cc)

```
void TigerParser::parse_() {
    std::string* fn = boost::get<std::string>(&input_);
    misc::symbol filename(fn == nullptr ? ""
                               : *fn == "-" ? "standard input" : *fn);
    location_.initialize(&filename.name_get());
    std::shared_ptr<std::istream> in;
    if (fn_ == "-")
        in.reset(&std::cin, [](...){});
    else {
        in = std::make_shared<std::ifstream>(filename);
        // Check for errors...
    }
    scanner_>scan_open(*in);
    parser parser(*this);
    parser.set_debug_level(parse_trace_p_);
    decs_ = nullptr; exp_ = nullptr;
    parser.parse();
    scanner_>scan_close();
}
```

The Parser (parse/parsetiger.yy)

```
%define filename_type {const std::string}  
%locations  
  
// The parsing context.  
%param { parse::TigerParser& tp }
```

Two Grammars in One

- 1 Flex & Bison: Recalls
- 2 Semantic Values
- 3 Locations
- 4 Improving the Scanner/Parser
 - Error Recovery
 - Pure Parser
 - Two Grammars in One
 - Reentrancy
- 5 Symbols

The Parser

parse/parsetiger.yy

```
%token SEED_IMPORT  "seed-import"
%token SEED_SOURCE   "seed-source"
%%
program:
    /* Parsing a source program. */
    "seed-source" exp          { tp.exp_ = $2; }
| /* Parsing an imported file. */
  "seed-import" "let" decs "end" { tp.decs_ = $3; }
;
```

The Scanner: Wrapping yyflex

```
parse/scantiger.ll
```

```
int
yylex (yystype *yylval, yy::location *yyloc,
       parse::TigerParser& tp)
{
    if (tp.seed_)
    {
        int res = 0;
        std::swap(res, tp.seed_);
        return res;
    }
    else
        return flex_yylex(yylval, yyloc, tp);
}
```

The Scanner: Using the top of yyflex

```
parse/scantiger.ll
```

```
%%  
%{  
    if (tp.seed_)  
    {  
        int res = 0;  
        std::swap(res, tp.seed_);  
        return res;  
    }  
%}
```

Without Seeds

parse/parsetiger.yy

```
%%  
program:  
    /* Parsing a source program. */  
    exp    { tp.exp_ = $1; }  
| /* Parsing an imported file. */  
    decs   { tp.decs_ = $1; }  
;
```

Reentrancy

- 1 Flex & Bison: Recalls
- 2 Semantic Values
- 3 Locations
- 4 Improving the Scanner/Parser**
 - Error Recovery
 - Pure Parser
 - Two Grammars in One
 - **Reentrancy**
- 5 Symbols


```
parse/scantiger.ll
```

```
void yyFlexLexer::scan_open_(std::istream& f)
{
    yypush_buffer_state(YY_CURRENT_BUFFER);
    yy_switch_to_buffer(yy_create_buffer(&f, YY_BUF_SIZE));
}

void yyFlexLexer::scan_close_()
{
    yypop_buffer_state();
}
```

Recursive Invocation of the Parser

parse/parsetiger.yy

```
importdec: "import" STRING
{
    $$ = tp.parse_import(take($2), @$);
    // Parsing may have failed.
    if (!$$)
        $$ = new ast::decs_list_type;
}
;
```

Symbols

- 1 Flex & Bison: Recalls
- 2 Semantic Values
- 3 Locations
- 4 Improving the Scanner/Parser
- 5 Symbols**
 - cstats
 - Symbols

- 1 Flex & Bison: Recalls
- 2 Semantic Values
- 3 Locations
- 4 Improving the Scanner/Parser
- 5 Symbols**
 - **cstats**
 - Symbols

cstats: Counting Symbols

```
g++ -E -P "$@" \  
| tr -cs '[:alnum:]_\' '\[\n*]\'' \  
| grep '^[[alpha:]]\' \  
| grep -v -E -w "$cxx_keywords" > $tmp.1  
total=$(wc -lc < $tmp.1 \  
| awk '{print $1 " (" $2 " chars)"}')  
sort $tmp.1 \  
| uniq -c \  
| sed 's/^  //;s/\t/ /' \  
| sort -rn >$tmp.2  
unique=$(sed -s 's/.* //' $tmp.2 | wc -lc \  
| awk '{print $1 " (" $2 " chars)"}')  
echo $total occurrences of $unique symbols.  
sed 42q $tmp.2 \  
| pr --page-width=60 --column=3 --omit-header  
rm -f $tmp.*
```

Lemon (as-of 2019-01-15)

15182 (78642 chars) occurrences of 1082 (8875 chars) symbols.

1868 gt	176 lineno	87 rule
943 quot	155 lt	87 h
654 i	149 cp	82 np
458 amp	148 s	78 filename
373 lemp	146 name	72 z
347 rp	139 cfp	71 fp
306 n	116 next	70 array
297 psp	109 stp	69 ht
227 fprintf	108 p	69 config
199 sp	107 a	62 errorcnt
198 out	101 type	62 action
187 j	94 state	61 lem
182 x	91 symbol	60 d
177 ap	89 c	56 data

GCC's C Parser

18958 (198353 chars)	occurrences of	5835 (89396 chars)	symbols.
2676 tree	89 new_type_flag	38 build_nt	
1579 ttype	70 cpp_reader	36 itype	
1123 yyvsp	69 build_tree_lis	36 build_x_binary	
909 yyval	67 parse	35 yychar	
358 ftype	65 y	35 frob_opname	
247 t	61 obstack	35 d	
206 gt_pointer_ope	58 GTY	34 e	
200 common	46 identifier	33 tree_code_type	
192 size_t	43 error	33 operator_name_	
175 code	40 cp_global_tree	33 C	
171 tree_code	39 yyn	32 got_scope	
123 FILE	39 s	31 IDENTIFIER_NOD	
97 rtx	39 lookups	30 tree_class_che	
95 type	38 TREE_LIST	30 global_trees	

Tiger Compiler's Driver (as-of 1.70)

8544 (83423 chars) occurrences of 1320 (16098 chars) symbols.

603 std	76 FILE	48 hash
354 size_t	74 false_type	47 iterator_trait
351 noexcept	73 declval	47 begin
334 size_type	71 reverse_iterat	46 compare
274 basic_string	64 difference_typ	46 char_traits
268 type	62 pointer	42 integral_const
202 constexpr	61 pair	41 allocator
158 char_type	56 int_type	40 C
153 forward	55 locale_t	39 first
114 value	53 value_type	37 string
96 decltype	53 move_iterator	37 replace
94 true_type	52 move	37 basic_istream
80 size	50 traits_type	36 exception_ptr
77 base	48 length	35 wstring

Symbols

- 1 Flex & Bison: Recalls
- 2 Semantic Values
- 3 Locations
- 4 Improving the Scanner/Parser
- 5 Symbols**
 - cstats
 - Symbols

Save Time and Space

One unique occurrence for each identifier:

In C a simple `const char*`

In C++ an iterator in a `std::set`

“Set has the important property that inserting a new element into a set does not invalidate iterators that point to existing elements.”

Save space fewer allocations

Save time fewer allocations, easier comparisons

Save nerves easier memory management

Save Time and Space

One unique occurrence for each identifier:

In C a simple `const char*`

In C++ an iterator in a `std::set`

“Set has the important property that inserting a new element into a set does not invalidate iterators that point to existing elements.”

Save space fewer allocations

Save time fewer allocations, easier comparisons

Save nerves easier memory management

Save Time and Space

One unique occurrence for each identifier:

In C a simple `const char*`

In C++ an iterator in a `std::set`

“Set has the important property that inserting a new element into a set does not invalidate iterators that point to existing elements.”

Save space fewer allocations

Save time fewer allocations, easier comparisons

Save nerves easier memory management

Save Time and Space

One unique occurrence for each identifier:

In C a simple `const char*`

In C++ an iterator in a `std::set`

“Set has the important property that inserting a new element into a set does not invalidate iterators that point to existing elements.”

Save space fewer allocations

Save time fewer allocations, easier comparisons

Save nerves easier memory management

Save Time and Space

One unique occurrence for each identifier:

In C a simple `const char*`

In C++ an iterator in a `std::set`

“Set has the important property that inserting a new element into a set does not invalidate iterators that point to existing elements.”

Save space fewer allocations

Save time fewer allocations, easier comparisons

Save nerves easier memory management

Save Time and Space

One unique occurrence for each identifier:

In C a simple `const char*`

In C++ an iterator in a `std::set`

“Set has the important property that inserting a new element into a set does not invalidate iterators that point to existing elements.”

Save space fewer allocations

Save time fewer allocations, easier comparisons

Save nerves easier memory management

Development Tools

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

EPITA — École Pour l'Informatique et les Techniques Avancées

February 3, 2020

Development Tools

- 1 Tools for the Developer
- 2 Maintaining Packages
- 3 tc Tasks

Tools for the Developer

- 1 Tools for the Developer
- 2 Maintaining Packages
- 3 tc Tasks

Other developing tools

- Use warnings
- Use the assert macro
- Use static_assert (C++11)
- Electric Fence, DUMA, PageHeap (Windows), Guard Malloc (Mac OS X)
- Dmalloc
- AddressSanitizer, ThreadSanitizer, Memory Sanitizer, Leaks Sanitizer
- Icov
- Mudflap (built in GCC 3.x–4.8)
- GDB (GUIs : DDD, KDbg), LLDB
- Purify, Insure++, BoundsChecker (proprietary)
- Valgrind
- Pin
- DTrace
- Clang Static Analyzer (LLVM)
- Cppcheck, crazy flint
- Gprof, OProfile
- clang tidy, clang format

Other developing tools

- Use warnings
- Use the `assert` macro
- Use `static_assert` (C++11)
- Electric Fence, DUMA, PageHeap (Windows), Guard Malloc (Mac OS X)
- Dmalloc
- AddressSanitizer, ThreadSanitizer, Memory Sanitizer, Leaks Sanitizer
- Icov
- Mudflap (built in GCC 3.x–4.8)
- GDB (GUIs : DDD, KDbg), LLDB
- Purify, Insure++, BoundsChecker (proprietary)
- Valgrind
- Pin
- DTrace
- Clang Static Analyzer (LLVM)
- Cppcheck, clazy, flint
- Gprof, OProfile
- clang tidy, clang format

Other developing tools

- Use warnings
- Use the `assert` macro
- Use `static_assert` (C++11)
- Electric Fence, DUMA, PageHeap (Windows), Guard Malloc (Mac OS X)
- Dmalloc
- AddressSanitizer, ThreadSanitizer, Memory Sanitizer, Leaks Sanitizer
- Icov
- Mudflap (built in GCC 3.x–4.8)
- GDB (GUIs : DDD, KDbg), LLDB
- Purify, Insure++, BoundsChecker (proprietary)
- Valgrind
- Pin
- DTrace
- Clang Static Analyzer (LLVM)
- Cppcheck, clazy, flint
- Gprof, OProfile
- clang tidy, clang format

Other developing tools

- Use warnings
- Use the `assert` macro
- Use `static_assert` (C++11)
- Electric Fence, DUMA, PageHeap (Windows), Guard Malloc (Mac OS X)
- Dmalloc
- AddressSanitizer, ThreadSanitizer, Memory Sanitizer, Leaks Sanitizer
- Icov
- Mudflap (built in GCC 3.x–4.8)
- GDB (GUIs : DDD, KDbg), LLDB
- Purify, Insure++, BoundsChecker (proprietary)
- Valgrind
- Pin
- DTrace
- Clang Static Analyzer (LLVM)
- Cppcheck, clazy, flint
- Gprof, OProfile
- clang tidy, clang format

Other developing tools

- Use warnings
- Use the assert macro
- Use static_assert (C++11)
- Electric Fence, DUMA, PageHeap (Windows), Guard Malloc (Mac OS X)
- Dmalloc
- AddressSanitizer, ThreadSanitizer, Memory Sanitizer, Leaks Sanitizer
- Icov
- Mudflap (built in GCC 3.x–4.8)
- GDB (GUIs : DDD, KDbg), LLDB
- Purify, Insure++, BoundsChecker (proprietary)
- Valgrind
- Pin
- DTrace
- Clang Static Analyzer (LLVM)
- Cppcheck, clazy flint
- Gprof, OProfile
- clang tidy, clang format

Other developing tools

- Use warnings
- Use the assert macro
- Use static_assert (C++11)
- Electric Fence, DUMA, PageHeap (Windows), Guard Malloc (Mac OS X)
- Dmalloc
- AddressSanitizer, ThreadSanitizer, Memory Sanitizer, Leaks Sanitizer
- lcov
- Mudflap (built in GCC 3.x–4.8)
- GDB (GUIs : DDD, KDbg), LLDB
- Purify, Insure++, BoundsChecker (proprietary)
- Valgrind
- Pin
- DTrace
- Clang Static Analyzer (LLVM)
- Cppcheck, clazy flint
- Gprof, OProfile
- clang tidy, clang format

Other developing tools

- Use warnings
- Use the `assert` macro
- Use `static_assert` (C++11)
- Electric Fence, DUMA, PageHeap (Windows), Guard Malloc (Mac OS X)
- Dmalloc
- AddressSanitizer, ThreadSanitizer, Memory Sanitizer, Leaks Sanitizer
- lcov
- Mudflap (built in GCC 3.x–4.8)
- GDB (GUIs : DDD, KDbg), LLDB
- Purify, Insure++, BoundsChecker (proprietary)
- Valgrind
- Pin
- DTrace
- Clang Static Analyzer (LLVM)
- Cppcheck, clazy flint
- Gprof, OProfile
- clang tidy, clang format

Other developing tools

- Use warnings
- Use the `assert` macro
- Use `static_assert` (C++11)
- Electric Fence, DUMA, PageHeap (Windows), Guard Malloc (Mac OS X)
- Dmalloc
- AddressSanitizer, ThreadSanitizer, Memory Sanitizer, Leaks Sanitizer
- Icov
- Mudflap (built in GCC 3.x–4.8)
- GDB (GUIs : DDD, KDbg), LLDB
- Purify, Insure++, BoundsChecker (proprietary)
- Valgrind
- Pin
- DTrace
- Clang Static Analyzer (LLVM)
- Cppcheck, clazy, flint
- Gprof, OProfile
- clang tidy, clang format

Other developing tools

- Use warnings
- Use the assert macro
- Use static_assert (C++11)
- Electric Fence, DUMA, PageHeap (Windows), Guard Malloc (Mac OS X)
- Dmalloc
- AddressSanitizer, ThreadSanitizer, Memory Sanitizer, Leaks Sanitizer
- Icov
- Mudflap (built in GCC 3.x–4.8)
- GDB (GUIs : DDD, KDbg), LLDB
- Purify, Insure++, BoundsChecker (proprietary)
- Valgrind
- Pin
- DTrace
- Clang Static Analyzer (LLVM)
- Cppcheck, clazy flint
- Gprof, OProfile
- clang tidy, clang format

Other developing tools

- Use warnings
- Use the assert macro
- Use static_assert (C++11)
- Electric Fence, DUMA, PageHeap (Windows), Guard Malloc (Mac OS X)
- Dmalloc
- AddressSanitizer, ThreadSanitizer, Memory Sanitizer, Leaks Sanitizer
- Icov
- Mudflap (built in GCC 3.x–4.8)
- GDB (GUIs : DDD, KDbg), LLDB
- Purify, Insure++, BoundsChecker (proprietary)
- Valgrind
- Pin
- DTrace
- Clang Static Analyzer (LLVM)
- Cppcheck, clazy flint
- Gprof, OProfile
- clang tidy, clang format

Other developing tools

- Use warnings
- Use the assert macro
- Use static_assert (C++11)
- Electric Fence, DUMA, PageHeap (Windows), Guard Malloc (Mac OS X)
- Dmalloc
- AddressSanitizer, ThreadSanitizer, Memory Sanitizer, Leaks Sanitizer
- Icov
- Mudflap (built in GCC 3.x–4.8)
- GDB (GUIs : DDD, KDbg), LLDB
- Purify, Insure++, BoundsChecker (proprietary)
- Valgrind
- Pin
- DTrace
- Clang Static Analyzer (LLVM)
- Cppcheck, clazy flint
- Gprof, OProfile
- clang tidy, clang format

Other developing tools

- Use warnings
- Use the assert macro
- Use static_assert (C++11)
- Electric Fence, DUMA, PageHeap (Windows), Guard Malloc (Mac OS X)
- Dmalloc
- AddressSanitizer, ThreadSanitizer, Memory Sanitizer, Leaks Sanitizer
- Icov
- Mudflap (built in GCC 3.x–4.8)
- GDB (GUIs : DDD, KDbg), LLDB
- Purify, Insure++, BoundsChecker (proprietary)
- Valgrind
- Pin
- DTrace
- Clang Static Analyzer (LLVM)
- Cppcheck, clazy flint
- Gprof, OProfile
- clang tidy, clang format

Other developing tools

- Use warnings
- Use the assert macro
- Use static_assert (C++11)
- Electric Fence, DUMA, PageHeap (Windows), Guard Malloc (Mac OS X)
- Dmalloc
- AddressSanitizer, ThreadSanitizer, Memory Sanitizer, Leaks Sanitizer
- lcov
- Mudflap (built in GCC 3.x–4.8)
- GDB (GUIs : DDD, KDbg), LLDB
- Purify, Insure++, BoundsChecker (proprietary)
- Valgrind
- Pin
- DTrace
- Clang Static Analyzer (LLVM)
- Cppcheck, clazy flint
- Gprof, OProfile
- clang tidy, clang format

Other developing tools

- Use warnings
- Use the assert macro
- Use static_assert (C++11)
- Electric Fence, DUMA, PageHeap (Windows), Guard Malloc (Mac OS X)
- Dmalloc
- AddressSanitizer, ThreadSanitizer, Memory Sanitizer, Leaks Sanitizer
- Icov
- Mudflap (built in GCC 3.x–4.8)
- GDB (GUIs : DDD, KDbg), LLDB
- Purify, Insure++, BoundsChecker (proprietary)
- Valgrind
- Pin
- DTrace
- Clang Static Analyzer (LLVM)
- Cppcheck, clazy flint
- Gprof, OProfile
- clang tidy, clang format

Other developing tools

- Use warnings
- Use the assert macro
- Use static_assert (C++11)
- Electric Fence, DUMA, PageHeap (Windows), Guard Malloc (Mac OS X)
- Dmalloc
- AddressSanitizer, ThreadSanitizer, Memory Sanitizer, Leaks Sanitizer
- Icov
- Mudflap (built in GCC 3.x–4.8)
- GDB (GUIs : DDD, KDbg), LLDB
- Purify, Insure++, BoundsChecker (proprietary)
- Valgrind
- Pin
- DTrace
- Clang Static Analyzer (LLVM)
- Cppcheck, clazy, flint
- Gprof, OProfile
- clang tidy, clang format

Other developing tools

- Use warnings
- Use the assert macro
- Use static_assert (C++11)
- Electric Fence, DUMA, PageHeap (Windows), Guard Malloc (Mac OS X)
- Dmalloc
- AddressSanitizer, ThreadSanitizer, Memory Sanitizer, Leaks Sanitizer
- Icov
- Mudflap (built in GCC 3.x–4.8)
- GDB (GUIs : DDD, KDbg), LLDB
- Purify, Insure++, BoundsChecker (proprietary)
- Valgrind
- Pin
- DTrace
- Clang Static Analyzer (LLVM)
- Cppcheck, crazy flint
- Gprof, OProfile
- clang tidy, clang format

Other developing tools

- Use warnings
- Use the assert macro
- Use static_assert (C++11)
- Electric Fence, DUMA, PageHeap (Windows), Guard Malloc (Mac OS X)
- Dmalloc
- AddressSanitizer, ThreadSanitizer, Memory Sanitizer, Leaks Sanitizer
- lcov
- Mudflap (built in GCC 3.x–4.8)
- GDB (GUIs : DDD, KDbg), LLDB
- Purify, Insure++, BoundsChecker (proprietary)
- Valgrind
- Pin
- DTrace
- Clang Static Analyzer (LLVM)
- Cppcheck, crazy flint
- Gprof, OProfile
- clang tidy, clang format

Bound Checking cppcheck

```
int main()
{
    int arr[ 2 ];
    cout << arr[ 4 ] << endl;
    return 0;
}
```

```
$ cppcheck foo.cc
```

```
Checking foo.cc ...
```

```
[foo.cc:8]: (error) Array 'arr[2]' accessed at index 4, which is out
```

```
[foo.cc:8]: (error) Uninitialized variable: arr
```

Bound Checking -fsanitize=bounds

```
int main(){  
    int tab[10];  
    int i;  
  
    for (i = 0; i <= 10; ++i)  
        tab[i] = 0;  
    return 0;  
}
```

```
$ g++ -fsanitize=bounds foo.cc
```

```
$ ./a.out
```

```
runtime error: index 10 out of bounds for type 'int [10]'
```

Address Sanitizer

```
int main()
{
    int arr[ 2 ];
    cout << arr[ 4 ] << endl;
    return 0;
}
```

```
gcc -fsanitize=address bounds-violation.c
```

```

=====
==8359== ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7fff23839248 at pc 0x400730 bp 0x7fff23839
WRITE of size 4 at 0x7fff23839248 thread T0
    #0 0x40072f (/work/roland/src/cours-ccmp/ccmp/dev-tools/a.out+0x40072f)
    #1 0x7f5c1b107eac (/lib/x86_64-linux-gnu/libc-2.13.so+0x10eac)
    #2 0x4005b8 (/work/roland/src/cours-ccmp/ccmp/dev-tools/a.out+0x4005b8)
Address 0x7fff23839248 is located at offset 72 in frame <main> of T0's stack:
    This frame has 1 object(s):
        [32, 72) 'tab'
HINT: this may be a false positive if your program uses some custom stack unwind mechanism or swapcontext
(longjmp and C++ exceptions *are* supported)
Shadow bytes around the buggy address:
    0x1000646ff1f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    0x1000646ff200: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    0x1000646ff210: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    0x1000646ff220: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    0x1000646ff230: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x1000646ff240: f1 f1 f1 f1 00 00 00 00 00[f4]f4 f4 f3 f3 f3 f3
    0x1000646ff250: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    0x1000646ff260: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    0x1000646ff270: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    0x1000646ff280: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    0x1000646ff290: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Shadow byte legend (one shadow byte represents 8 application bytes):
Addressable:           00
Partially addressable: 01 02 03 04 05 06 07
Heap left redzone:     fa
Heap right redzone:    fb
Freed Heap region:     fd
Stack left redzone:    f1
Stack mid redzone:     f2
Stack right redzone:   f3
Stack partial redzone: f4
Stack after return:    f5
Stack use after scope: f6
Global redzone:        f9
Global init order:     f6
Disowned by user:      f7

```

Mudflap

```
int main()
{
    int arr[ 2 ];
    cout << arr[ 4 ] << endl;
    return 0;
}
gcc -fmudflap -lmudflap bounds-violation.c
```


Mudflap

```
env MUDFLAP_OPTIONS=-viol-abort ./a.out
```

```
*****
```

```
mudflap violation 1 (check/write): time=1292501299.526454 ptr=0xbfc35d34 size=44  
pc=0xb77d13bd location='bounds-violation.c:8:5 (main)'
```

```
  /usr/lib/libmudflap.so.0(__mf_check+0x3d) [0xb77d13bd]
```

```
  ./a.out(main+0xb7) [0x804883b]
```

```
  /usr/lib/libmudflap.so.0(__wrap_main+0x49) [0xb77d0b89]
```

```
Nearby object 1: checked region begins 0B into and ends 4B after
```

```
mudflap object 0x8c7a080: name='bounds-violation.c:4:7 (main) tab'
```

```
bounds=[0xbfc35d34,0xbfc35d5b] size=40 area=stack check=0r/4w liveness=4
```

```
alloc time=1292501299.526444 pc=0xb77d0b2d
```

```
number of nearby objects: 1
```

```
zsh: abort      env MUDFLAP_OPTIONS=-viol-abort ./a.out
```

Valgrind and Memory Violation

```
#include <stdio.h>

typedef struct list_s { int val; struct list_s *next; } list_t;

list_t *list_new(int val, list_t *next) {
    list_t res = { val, next };
    return &res;
}

void list_print(const list_t *const list) {
    if (list)
        printf("%d\n", list->val), list_print(list->next);
}

int main(void) {
    list_print(list_new(2, list_new(1, list_new(0, NULL))));
    return 0;
}
```

Valgrind and Memory Leaks

```
#include <stdio.h>
#include <stdlib.h>

typedef struct list_s { int val; struct list_s *next; } list_t;

list_t *list_new(int val, list_t *next) {
    list_t *res = (list_t *) malloc(sizeof(list_t));
    res->val = val; res->next = next;
    return res;
}

void list_print(const list_t *const list) {
    if (list)
        printf("%d\n", list->val), list_print(list->next);
}

int main(void) {
    list_print(list_new(2, list_new(1, list_new(0, NULL))));
    return 0;
}
```

Valgrind and Memory Leaks

```
gcc -g memory-leaks.c
valgrind --leak-check=full ./a.out
==9702== Memcheck, a memory error detector
==9702== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==9702== Using Valgrind-3.6.0.SVN-Debian and LibVEX; rerun with -h for copyright info
==9702== Command: ./a.out
==9702==
2
1
0
==9702==
==9702== HEAP SUMMARY:
==9702==    in use at exit: 24 bytes in 3 blocks
==9702==    total heap usage: 3 allocs, 0 frees, 24 bytes allocated
==9702==
==9702== 24 (8 direct, 16 indirect) bytes in 1 blocks are definitely lost in loss record 3 of 3
==9702==    at 0x4023F50: malloc (vg_replace_malloc.c:236)
==9702==    by 0x8048405: list_new (memory-leaks.c:7)
==9702==    by 0x804848D: main (memory-leaks.c:18)
==9702==
==9702== LEAK SUMMARY:
==9702==    definitely lost: 8 bytes in 1 blocks
==9702==    indirectly lost: 16 bytes in 2 blocks
==9702==    possibly lost: 0 bytes in 0 blocks
==9702==    still reachable: 0 bytes in 0 blocks
==9702==         suppressed: 0 bytes in 0 blocks
==9702==
==9702== For counts of detected and suppressed errors, rerun with: -v
==9702== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 11 from 6)
```

A Clear Winner?

- Valgrind doesn't catch the previous Mudflap example.
 - Padding
 - Overrun into neighbor regions
- Mudflap and ASan do not know about uninitialized regions.

A Clear Winner?

- Valgrind doesn't catch the previous Mudflap example.
 - Padding
 - Overrun into neighbor regions
- Mudflap and ASan do not know about uninitialized regions.

A Clear Winner?

- Valgrind doesn't catch the previous Mudflap example.
 - Padding
 - Overrun into neighbor regions
- Mudflap and ASan do not know about uninitialized regions.

A Clear Winner?

- Valgrind doesn't catch the previous Mudflap example.
 - Padding
 - Overrun into neighbor regions
- Mudflap and ASan do not know about uninitialized regions.

A Clear Winner? Kind of!

Actually Valgrind *does* catch the previous Mudflap example now, thanks to SGCheck, a stack and global array overrun detector leveraging debugging information.

```
gcc -g bounds-violation.c  
valgrind --tool=exp-sgcheck ./a.out
```

```
=22757= exp-sgcheck, a stack and global array overrun detector  
=22757= NOTE: This is an Experimental-Class Valgrind Tool  
=22757= Copyright (C) 2003-2013, and GNU GPL'd, by OpenWorks Ltd et al.  
=22757= Using Valgrind-3.9.0 and LibVEX; rerun with -h for copyright info  
=22757= Command: ./a.out  
=22757=  
=22757= Invalid write of size 4  
=22757=   at 0x4004BE: main (bounds-violation.c:8)  
=22757=   Address 0xffff000408 expected vs actual:  
=22757=   Expected: stack array "tab" of size 40 in this frame  
=22757=   Actual:   unknown  
=22757=   Actual:   is 0 after Expected  
=22757=  
=22757=  
=22757= ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 4 from 4)
```

Intermission: `goto` Still Considered Harmful

Old (bad!) habits die hard: Apple's Feb. 2014 SSL/TLS Bug [Langley, 2014]

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
                                uint8_t *signature, UInt16 signatureLen)
{
    OSStatus      err;
    ...

    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
    ...

fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
}
```

Intermission: `goto` Still Considered Harmful

Old (bad!) habits die hard: Apple's Feb. 2014 SSL/TLS Bug
[Langley, 2014]

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
                                uint8_t *signature, UInt16 signatureLen)
{
    OSStatus      err;
    ...

    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
    goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0) // Unreachable code.
        goto fail;
    ...

fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
}
```

Caught by Clang's `-Wunreachable-code` flag (but not `-Wall`)

- Makes working in group *a lot* easier.
- Gives the possibility to travel back in time (e.g, to hunt bugs).
- Allows several, non-linear developing models (branches).
- Add some semantics to the development itself.
- Provides a kind of backup
But cannot make up for the lack of a real backup solution!
- EPITA provides Git repositories for the Tiger project.

- Makes working in group *a lot* easier.
- Gives the possibility to travel back in time (e.g, to hunt bugs).
- Allows several, non-linear developing models (branches).
- Add some semantics to the development itself.
- Provides a kind of backup
But cannot make up for the lack of a real backup solution!
- EPITA provides Git repositories for the Tiger project.

- Makes working in group *a lot* easier.
- Gives the possibility to travel back in time (e.g, to hunt bugs).
- Allows several, non-linear developing models (branches).
- Add some semantics to the development itself.
- Provides a kind of backup
But cannot make up for the lack of a real backup solution!
- EPITA provides Git repositories for the Tiger project.

- Makes working in group *a lot* easier.
- Gives the possibility to travel back in time (e.g, to hunt bugs).
- Allows several, non-linear developing models (branches).
- Add some semantics to the development itself.
- Provides a kind of backup
But cannot make up for the lack of a real backup solution!
- EPITA provides Git repositories for the Tiger project.

- Makes working in group *a lot* easier.
- Gives the possibility to travel back in time (e.g, to hunt bugs).
- Allows several, non-linear developing models (branches).
- Add some semantics to the development itself.
- Provides a kind of backup
But cannot make up for the lack of a real backup solution!
- EPITA provides Git repositories for the Tiger project.

- Makes working in group *a lot* easier.
- Gives the possibility to travel back in time (e.g, to hunt bugs).
- Allows several, non-linear developing models (branches).
- Add some semantics to the development itself.
- Provides a kind of backup
But cannot make up for the lack of a real backup solution!
- EPITA provides Git repositories for the Tiger project.

- Use comments to annotate code entities (namespaces, files, functions, classes, type aliases, etc.).
- Generate a hyper-text reference documentation.
- Several back-ends: HTML, PDF, RTF, etc.
- Use `make doc` in the Tiger Project.
- See <http://doxygen.org/manual.html>.

- Use comments to annotate code entities (namespaces, files, functions, classes, type aliases, etc.).
- Generate a hyper-text reference documentation.
- Several back-ends: HTML, PDF, RTF, etc.
- Use `make doc` in the Tiger Project.
- See <http://doxygen.org/manual.html>.

Document with Doxygen

- Use comments to annotate code entities (namespaces, files, functions, classes, type aliases, etc.).
- Generate a hyper-text reference documentation.
- Several back-ends: HTML, PDF, RTF, etc.
- Use `make doc` in the Tiger Project.
- See <http://doxygen.org/manual.html>.

Document with Doxygen

- Use comments to annotate code entities (namespaces, files, functions, classes, type aliases, etc.).
- Generate a hyper-text reference documentation.
- Several back-ends: HTML, PDF, RTF, etc.
- Use `make doc` in the Tiger Project.
- See <http://doxygen.org/manual.html>.

- Use comments to annotate code entities (namespaces, files, functions, classes, type aliases, etc.).
- Generate a hyper-text reference documentation.
- Several back-ends: HTML, PDF, RTF, etc.
- Use `make doc` in the Tiger Project.
- See <http://doxygen.org/manual.html>.

Document with Doxygen: type/libtype.hh

```
/// \file type/libtype.hh
/// \brief Declare the function exported by type module.
#ifndef TYPE_LIBTYPE_HH
# define TYPE_LIBTYPE_HH

# include "misc/error.hh"
# include "ast/fwd.hh"

/// Type-checking an ast::Ast.
namespace type
{

    /** \brief Check types in a (bound) AST.
     ** \param tree    abstract syntax tree's root.
     ** \return        synthesis of the errors possibly found. */
    misc::error types_check(ast::Ast& tree);

} // namespace type
```

Maintaining Packages

1 Tools for the Developer

2 Maintaining Packages

- GNU Tools
- Autoconf for tc
- Automake for tc

3 tc Tasks

1 Tools for the Developer

2 Maintaining Packages

- GNU Tools

- Autoconf for tc

- Automake for tc

3 tc Tasks

aclocal Create `aclocal.m4` from `configure.ac`'s requests

`autoconf` Create `configure` from `configure.ac` and `aclocal.m4`

`autoheader` Create `config.h.in` from `configure.ac` (and `aclocal.m4`)

`automake` Create `Makefile.in` from `Makefile.am` and `configure.ac`

`autoreconf` Run them as needed (`autoreconf -fivm`)

Read Alexandre Duret-Lutz's Tutorial [Duret-Lutz, 2006]

aclocal Create `aclocal.m4` from `configure.ac`'s requests

autoconf Create `configure` from `configure.ac` and `aclocal.m4`

autoheader Create `config.h.in` from `configure.ac` (and `aclocal.m4`)

automake Create `Makefile.in` from `Makefile.am` and `configure.ac`

autoreconf Run them as needed (`autoreconf -fivm`)

Read Alexandre Duret-Lutz's Tutorial [Duret-Lutz, 2006]

aclocal Create `aclocal.m4` from `configure.ac`'s requests

autoconf Create `configure` from `configure.ac` and `aclocal.m4`

autoheader Create `config.h.in` from `configure.ac` (and `aclocal.m4`)

automake Create `Makefile.in` from `Makefile.am` and `configure.ac`

autoreconf Run them as needed (`autoreconf -fivm`)

Read Alexandre Duret-Lutz's Tutorial [Duret-Lutz, 2006]

aclocal Create `aclocal.m4` from `configure.ac`'s requests

autoconf Create `configure` from `configure.ac` and `aclocal.m4`

autoheader Create `config.h.in` from `configure.ac` (and `aclocal.m4`)

automake Create `Makefile.in` from `Makefile.am` and `configure.ac`

autoreconf Run them as needed (`autoreconf -fivm`)

Read [Alexandre Duret-Lutz's Tutorial](#) [Duret-Lutz, 2006]

aclocal Create `aclocal.m4` from `configure.ac`'s requests

autoconf Create `configure` from `configure.ac` and `aclocal.m4`

autoheader Create `config.h.in` from `configure.ac` (and `aclocal.m4`)

automake Create `Makefile.in` from `Makefile.am` and `configure.ac`

autoreconf Run them as needed (`autoreconf -fivm`)

Read Alexandre Duret-Lutz's Tutorial [Duret-Lutz, 2006]

aclocal Create `aclocal.m4` from `configure.ac`'s requests

autoconf Create `configure` from `configure.ac` and `aclocal.m4`

autoheader Create `config.h.in` from `configure.ac` (and `aclocal.m4`)

automake Create `Makefile.in` from `Makefile.am` and `configure.ac`

autoreconf Run them as needed (`autoreconf -fivm`)

Read Alexandre Duret-Lutz's Tutorial [Duret-Lutz, 2006]

Programming Tools: Packages

A set of packages to maintain packages:

Autoconf package configuration [MacKenzie et al., 2003]

Automake package build [Duret-Lutz and Tromeu, 2003]

Libtool portable build of shared libs

Gettext package internationalization

Flex scanner generation

Bison parser generation

Boost C++²

Programming Tools: Packages

A set of packages to maintain packages:

Autoconf package configuration [MacKenzie et al., 2003]

Automake package build [Duret-Lutz and Tromeu, 2003]

Libtool portable build of shared libs

Gettext package internationalization

Flex scanner generation

Bison parser generation

Boost C++²

Programming Tools: Packages

A set of packages to maintain packages:

Autoconf package configuration [MacKenzie et al., 2003]

Automake package build [Duret-Lutz and Tromeu, 2003]

Libtool portable build of shared libs

Gettext package internationalization

Flex scanner generation

Bison parser generation

Boost C++²

Programming Tools: Packages

A set of packages to maintain packages:

Autoconf package configuration [MacKenzie et al., 2003]

Automake package build [Duret-Lutz and Tromeu, 2003]

Libtool portable build of shared libs

Gettext package internationalization

Flex scanner generation

Bison parser generation

Boost C++²

Programming Tools: Packages

A set of packages to maintain packages:

Autoconf package configuration [MacKenzie et al., 2003]

Automake package build [Duret-Lutz and Tromeu, 2003]

Libtool portable build of shared libs

Gettext package internationalization

Flex scanner generation

Bison parser generation

Boost C++²

Programming Tools: Packages

A set of packages to maintain packages:

Autoconf package configuration [MacKenzie et al., 2003]

Automake package build [Duret-Lutz and Tromeu, 2003]

Libtool portable build of shared libs

Gettext package internationalization

Flex scanner generation

Bison parser generation

Boost C++²

Programming Tools: Packages

A set of packages to maintain packages:

Autoconf package configuration [MacKenzie et al., 2003]

Automake package build [Duret-Lutz and Tromeu, 2003]

Libtool portable build of shared libs

Gettext package internationalization

Flex scanner generation

Bison parser generation

Boost C++²

Autoconf for tc

1 Tools for the Developer

2 Maintaining Packages

- GNU Tools
- Autoconf for tc
- Automake for tc

3 tc Tasks

Configuring a package

```
configure -----+-----> config.log
                  |
config.h.in -.      v      .-> config.h -.
              +--> config.status -+      +--> make
Makefile.in -'                  '-> Makefile -'
```

Preparing a package for distribution

```
configure.ac --.
              |   .-----> autoconf ----> configure
              +----+
              |   '-----> autoheader --> config.h.in
aclocal.m4 ----'
```


Autoconf files

Configuring a package

```
configure -----+-----> config.log
                  |
config.h.in -.      v      .-> config.h -.
              +-> config.status -+      +--> make
Makefile.in -'              '-> Makefile -'
```

Preparing a package for distribution

```
configure.ac --.
              |   .-----> autoconf ----> configure
              +----+
              |   '------> autoheader --> config.h.in
aclocal.m4 ----'
```

configure.ac 1: Initialization

```
AC_PREREQ([2.64])
AC_INIT([LRDE Tiger Compiler], [1.72a],
        [tiger@lrde.epita.fr], [tc])

# Auxiliary files.
AC_CONFIG_AUX_DIR([build-aux])
AC_CONFIG_MACRO_DIR([build-aux/m4])

# Automake.
AM_INIT_AUTOMAKE([1.14.1 check-news dist-bzip2 no-dist-gzip
                  foreign
                  color-tests parallel-tests
                  nostdinc silent-rules -Wall])
AM_SILENT_RULES([yes])
```

configure.ac 2: C++ Compiler

```
# Look for a C++ compiler.
```

```
AC_LANG([C++])
```

```
AC_PROG_CXX
```

```
# Enable C++ 2017 support.
```

```
...
```

```
# Using pipes between compiler stages is faster.
```

```
AX_CHECK_COMPILE_FLAG([-pipe], [CXXFLAGS="$CXXFLAGS -pipe"])
```

```
# Use good warnings.
```

```
TC_CXX_WARNINGS([[ -Wall], [-W], [-Wcast-align], ...])
```

configure.ac 3: Auxiliary Programs

```
TC_PROG([flex], [>= 2.5.35], [FLEX],  
        [Flex scanner generator])  
AX_CONFIG_SCRIPTS([build-aux/bin/flex++])
```

```
TC_PROG([bison], [>= 3.2], [BISON],  
        [Bison parser generator])  
AX_CONFIG_SCRIPTS([build-aux/bin/bison++])
```

```
# We don't need static libraries, speed the compilation up.  
LT_INIT([disable-shared])
```

```
TC_PROG([monoburg], [>= 1.0.6], [MONOBURG],  
        [MonoBURG code generator generator])  
AX_CONFIG_SCRIPTS([build-aux/bin/monoburg++])
```

```
# Fix minimum version and then check it  
m4_define([TC_HAVM_PREREQ], [0.27])  
TC_PROG([havm], [>=TC_HAVM_PREREQ], [HAVM],  
        [The Tree Virtual Machine])
```

configure.ac 4: Libraries

```
AC_CONFIG_SUBDIRS([lib/argp])
```

```
BOOST_REQUIRE([1.63])
```

```
BOOST_CONVERSION # lexical_cast
```

```
BOOST_GRAPH
```

```
BOOST_PREPROCESSOR
```

```
BOOST_STRING_ALGO
```

```
BOOST_TRIBOOL
```

```
BOOST_VARIANT
```

configure.ac 5: SWIG & tcsh

```
TC_WITH_TCSH([with_tcsch=yes], [with_tcsch=no])  
AM_CONDITIONAL([ENABLE_TCSH], [test x$with_tcsch = xyes])  
  
AC_CONFIG_FILES([tcsch/Makefile  
                  tcsch/python/Makefile  
                  tcsch/ruby/Makefile])  
AX_CONFIG_SCRIPTS([tcsch/run])
```

configure.ac 6: File Creation

```
# Ask for the creation of config.h.  
AC_CONFIG_HEADERS([config.h])  
  
# Ask for the creation of the Makefiles.  
AC_CONFIG_FILES([Makefile])  
  
# Instantiate the output files.  
AC_OUTPUT
```

Automake for tc

1 Tools for the Developer

2 Maintaining Packages

- GNU Tools
- Autoconf for tc
- Automake for tc

3 tc Tasks

src/local.am 1: Variables

```
AUTOMAKE_OPTIONS = subdir-objects
```

```
AM_DEFAULT_SOURCE_EXT = .cc
```

```
BUILT_SOURCES =
```

```
CLEANFILES =
```

```
EXTRA_DIST =
```

```
MAINTAINERCLEANFILES =
```

```
TESTS =
```

```
EXTRA_PROGRAMS = $(TESTS)
```

```
dist_noinst_DATA =
```

```
noinst_LTLIBRARIES =
```

```
RECHECK_LOGS =
```

src/local.am 2: Common Options

```
# Most headers are to be shipped and to be found in src/, e.g.
# tasks/tasks.hh is shipped in $(top_srcdir)/src/task/tasks.hh.
# Some are *built* in src, e.g., $(top_builddir)/src/modules.hh.
AM_CPPFLAGS = -I$(top_srcdir)/lib
AM_CPPFLAGS += -I$(top_srcdir)/src -I$(top_builddir)/src
AM_CPPFLAGS += $(BOOST_CPPFLAGS)
# Find the prelude.
AM_CPPFLAGS += -DPKGDATADIR="\$(pkgdatadir)\\"
AM_CXXFLAGS = $(WARNING_CXXFLAGS)
```

src/local.am 3: Tasks

```
TASKS =  
include task/local.am  
include ast/local.am  
[...]  
include regalloc/local.am  
  
EXTRA_DIST += tiger_common.i
```

src/local.am 3: Building libtc

```
lib_LTLIBRARIES = src/libtc.la
src_libtc_la_SOURCES = src/version.hh
nodist_src_libtc_la_SOURCES = src/version.cc
src_libtc_la_LDFLAGS = $(BOOST_PROGRAM_OPTIONS_LDFLAGS)

src_libtc_la_LIBADD = \
    $(top_builddir)/lib/misc/libmisc.la \
    $(BOOST_PROGRAM_OPTIONS_LIBS)
```

src/local.am 4: Building tc

```
bin_PROGRAMS = src/tc
dist_src_tc_SOURCES =
    src/doc.hh
    $(TASKS)
    src/common.cc src/common.hh
    src/tc.cc

src_tc_LDADD = src/libtc.la
```

\
\
\
\

src/bind/local.am: Binding Names

```
## bind module.
```

```
EXTRA_DIST += %D%/tiger_bind.i
```

```
src_libtc_la_SOURCES += \
    %D%/binder.hh %D%/binder.hxx %D%/binder.cc \
    %D%/libbind.hh %D%/libbind.cc
```

```
TASKS += %D%/tasks.hh %D%/tasks.cc
```

```
# Tests.
```

```
check_PROGRAMS += %D%/test-bind
%C%_test_bind_LDADD = src/libtc.la
```

- 1 Tools for the Developer
- 2 Maintaining Packages
- 3 tc Tasks

The Tiger Compiler layout

- One module, one namespace
- One “library” per module, with a pure interface (`libfoo.*`)
- One task set per module, maybe impure (`tasks.*`)
- Tasks describe the command line interface
- Requirements over tasks order
- One class, one file base name:
 - `foo.hh` Interface
 - `foo.hxx` Inline implementation
 - `foo.cc` Implementation

The Tiger Compiler layout

- One module, one namespace
- One “library” per module, with a pure interface (`libfoo.*`)
- One task set per module, maybe impure (`tasks.*`)
- Tasks describe the command line interface
- Requirements over tasks order
- One class, one file base name:
 - `foo.hh` Interface
 - `foo.hxx` Inline implementation
 - `foo.cc` Implementation

The Tiger Compiler layout

- One module, one namespace
- One “library” per module, with a pure interface (`libfoo.*`)
- One task set per module, maybe impure (`tasks.*`)
- Tasks describe the command line interface
- Requirements over tasks order
- One class, one file base name:
 - `foo.hh` Interface
 - `foo.hxx` Inline implementation
 - `foo.cc` Implementation

The Tiger Compiler layout

- One module, one namespace
- One “library” per module, with a pure interface (`libfoo.*`)
- One task set per module, maybe impure (`tasks.*`)
- Tasks describe the command line interface
- Requirements over tasks order
- One class, one file base name:
 - `foo.hh` Interface
 - `foo.hxx` Inline implementation
 - `foo.cc` Implementation

The Tiger Compiler layout

- One module, one namespace
- One “library” per module, with a pure interface (`libfoo.*`)
- One task set per module, maybe impure (`tasks.*`)
- Tasks describe the command line interface
- Requirements over tasks order
- One class, one file base name:

`foo.hh` Interface

`foo.hxx` Inline implementation

`foo.cc` Implementation

The Tiger Compiler layout

- One module, one namespace
- One “library” per module, with a pure interface (`libfoo.*`)
- One task set per module, maybe impure (`tasks.*`)
- Tasks describe the command line interface
- Requirements over tasks order
- One class, one file base name:

`foo.hh` Interface

`foo.hxx` Inline implementation

`foo.cc` Implementation

The Tiger Compiler layout

- One module, one namespace
- One “library” per module, with a pure interface (`libfoo.*`)
- One task set per module, maybe impure (`tasks.*`)
- Tasks describe the command line interface
- Requirements over tasks order
- One class, one file base name:

`foo.hh` Interface

`foo.hxx` Inline implementation

`foo.cc` Implementation

The Tiger Compiler layout

- One module, one namespace
- One “library” per module, with a pure interface (`libfoo.*`)
- One task set per module, maybe impure (`tasks.*`)
- Tasks describe the command line interface
- Requirements over tasks order
- One class, one file base name:

`foo.hh` Interface

`foo.hxx` Inline implementation

`foo.cc` Implementation

The Tiger Compiler layout

- One module, one namespace
- One “library” per module, with a pure interface (`libfoo.*`)
- One task set per module, maybe impure (`tasks.*`)
- Tasks describe the command line interface
- Requirements over tasks order
- One class, one file base name:

`foo.hh` Interface

`foo.hxx` Inline implementation

`foo.cc` Implementation

Tasks: ast/tasks.hh

```
namespace ast
{
    namespace tasks
    {
        /// Global root node of abstract syntax tree.
        extern ast::DecsList* the_program;

        TASK_GROUP("2. Abstract Syntax Tree");

        /// Display the abstract syntax tree
        TASK_DECLARE("A|ast-display", "display the AST",
                    ast_display, "parse");

        /// Free the ast (if defined)
        TASK_DECLARE("D|ast-delete", "delete the AST",
                    ast_delete, "parse");
    } // namespace tasks
} // namespace ast
```

Tasks: ast/tasks.cc

```
namespace ast
{
    namespace tasks
    {
        ast::DecsList* the_program = nullptr;

        void ast_display()
        {
            precondition(the_program);
            std::cout << "/* Abstract Syntax Tree. */\n"
                      << *the_program << '\n';
        }

        void ast_delete()
        {
            delete the_program;
            the_program = nullptr;
        }
    } // namespace tasks
}
```

Bibliography I



Duret-Lutz, A. (2006).

The Autotools Tutorial.

<http://www-src.lip6.fr/homepages/Alexandre.Duret-Lutz/dl/autotools.pdf/>.



Duret-Lutz, A. and Tromeu, T. (2003).

GNU Automake.

<http://www.gnu.org/software/automake/>.



Langley, A. (2014).

ImperialViolet — Apple SSL/TLS bug.

<https://www.imperialviolet.org/2014/02/22/applebug.html>.



MacKenzie, D. J., Elliston, B., and Demaille, A. (2003).

GNU Autoconf.

<http://www.gnu.org/software/autoconf/>.

Abstract Syntax Trees

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

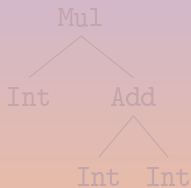
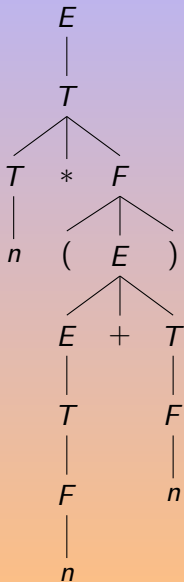
EPITA — École Pour l'Informatique et les Techniques Avancées

February 2, 2020

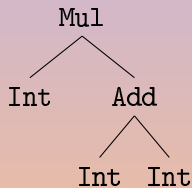
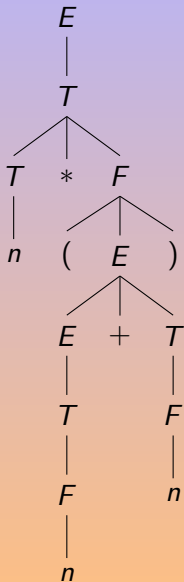
Abstract Syntax Trees

- 1 Structured Data for Input/Output: Trees
- 2 Algorithms on trees: Traversals
- 3 Applications
- 4 The Case of the Tiger Compiler

$1 * (2 + 3)$, 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

- 1 Structured Data for Input/Output: Trees
 - AST Generators
 - Exchanging Trees
 - Simple Implementation of ast in C++
- 2 Algorithms on trees: Traversals
- 3 Applications
- 4 The Case of the Tiger Compiler

AST Generators

- 1 Structured Data for Input/Output: Trees
 - AST Generators
 - Exchanging Trees
 - Simple Implementation of ast in C++
- 2 Algorithms on trees: Traversals
- 3 Applications
- 4 The Case of the Tiger Compiler

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

- 1 Structured Data for Input/Output: Trees
 - AST Generators
 - **Exchanging Trees**
 - Simple Implementation of ast in C++
- 2 Algorithms on trees: Traversals
- 3 Applications
- 4 The Case of the Tiger Compiler

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 [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 [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 [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

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

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

- Extensible mechanism for serializing data
- Language neutral
- Platform 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;  
}
```

```
// 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;
```


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"}`

- 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++

- 1 Structured Data for Input/Output: Trees
 - AST Generators
 - Exchanging Trees
 - Simple Implementation of ast in C++
- 2 Algorithms on trees: Traversals
- 3 Applications
- 4 The Case of the Tiger Compiler

Simple Grammar

Concrete Grammar (BNF)

```
<exp> ::= <exp> '+' <term>
        | <exp> '-' <term>
        | <term>.
<term> ::= <term> '*' <factor>
        | <term> '/' <factor>
        | <factor>.
<factor> ::= '(' <exp> ')',
           | <num>.
```

Abstract Grammar (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 Grammar (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

1 Structured Data for Input/Output: Trees

2 Algorithms on trees: Traversals

- Supporting the operator<<
- Multimethods
- Visitors
- Further with Visitors

3 Applications

4 The Case of the Tiger Compiler

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<<

1 Structured Data for Input/Output: Trees

2 Algorithms on trees: Traversals

- Supporting the operator<<
- Multimethods
- Visitors
- Further with Visitors

3 Applications

4 The Case of the Tiger Compiler

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;
}
```


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;
}
```

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?

1 Structured Data for Input/Output: Trees

2 Algorithms on trees: Traversals

- Supporting the operator<<
- **Multimethods**
- Visitors
- Further with Visitors

3 Applications

4 The Case of the Tiger Compiler

- 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

- 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

- 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

- 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

- 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

- 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

- 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 Hierarchy 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 Hierarchy 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 Hierarchy 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 Hierarchy to Dispatch

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

Ask the Hierarchy 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**.

1 Structured Data for Input/Output: Trees

2 Algorithms on trees: Traversals

- Supporting the operator<<
- Multimethods
- **Visitors**
- Further with Visitors

3 Applications

4 The Case of the Tiger Compiler

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_;
};
```

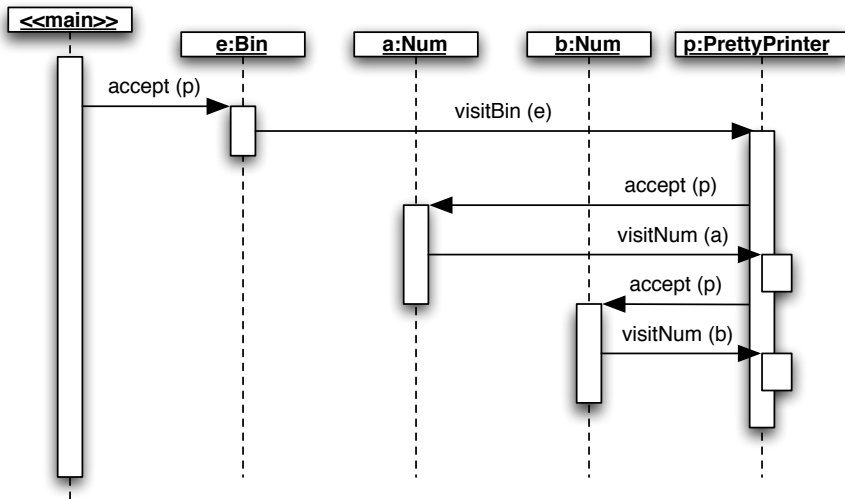
operator<< and main

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

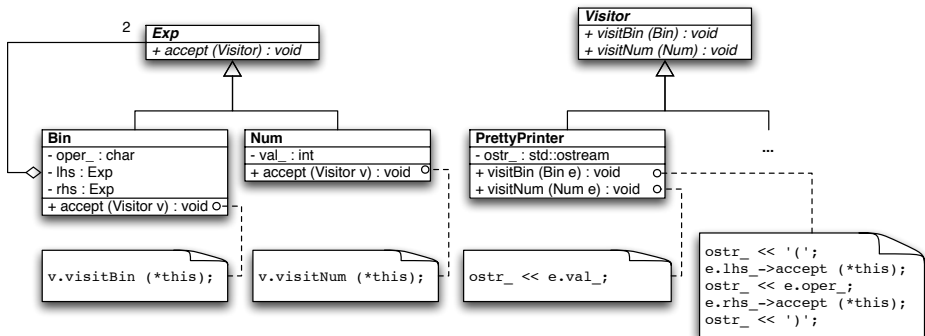
int
main()
{
    Bin* bin = new Bin('+', new Num(42), new Num(51));
    Exp* exp = bin;
    std::cout << "Bin: " << *bin << std::endl;
    std::cout << "Exp: " << *exp << std::endl;
    delete bin;
}
```

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

1 Structured Data for Input/Output: Trees

2 Algorithms on trees: Traversals

- Supporting the operator<<
- Multimethods
- Visitors
- Further with Visitors

3 Applications

4 The Case of the Tiger Compiler

- 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

- 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

- 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 Expr e) {  
    auto eval = Evaluator();
```

```
    return eval.eval(e);  
}
```

provided

```
int eval(const Expr e) {  
    auto eval = Evaluator();
```

```
    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 Expr e) {  
    auto eval = Evaluator();
```

```
    return eval(e);  
}
```

provided

```
int eval(const Expr e) {  
    auto eval = Evaluator();
```

```
    return eval(e);  
}
```

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

- 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();
    //...
}
```

- 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();
    //...
}
```

- 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)

- 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.

$\text{Twice}(v) := \text{Sequence}(v, v)$

$\text{Try}(v) := \text{Choice}(v, \text{Identity})$

$\text{TopDown}(v) := \text{Sequence}(v, \text{All}(\text{TopDown}(v)))$

$\text{BottomUp}(v) := \text{Sequence}(\text{All}(\text{BottomUp}(v)), v)$

Applications

- 1 Structured Data for Input/Output: Trees
- 2 Algorithms on trees: Traversals
- 3 Applications
 - Desugaring
 - Existing Tools
- 4 The Case of the Tiger Compiler

Desugaring

- 1 Structured Data for Input/Output: Trees
- 2 Algorithms on trees: Traversals
- 3 Applications
 - Desugaring
 - Existing Tools
- 4 The Case of the Tiger Compiler

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

- 1 Structured Data for Input/Output: Trees
- 2 Algorithms on trees: Traversals
- 3 Applications**
 - Desugaring
 - Existing Tools
- 4 The Case of the Tiger Compiler

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

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.

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.

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

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

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

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

- 1 Structured Data for Input/Output: Trees
- 2 Algorithms on trees: Traversals
- 3 Applications
- 4 The Case of the Tiger Compiler
 - The ast
 - Syntactic Sugar
 - Visitors

The ast

- 1 Structured Data for Input/Output: Trees
- 2 Algorithms on trees: Traversals
- 3 Applications
- 4 The Case of the Tiger Compiler
 - The ast
 - Syntactic Sugar
 - Visitors

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

- 1 Structured Data for Input/Output: Trees
- 2 Algorithms on trees: Traversals
- 3 Applications
- 4 The Case of the Tiger Compiler
 - The ast
 - Syntactic Sugar
 - Visitors

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

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 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 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 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

- 1 Structured Data for Input/Output: Trees
- 2 Algorithms on trees: Traversals
- 3 Applications
- 4 The Case of the Tiger Compiler
 - The ast
 - Syntactic Sugar
 - **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>
{
    // ...
};
```

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

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

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

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

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

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

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

`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

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

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

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

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

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

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

- `object::DesugarVisitor` Desugar Object Tiger code into the non-object core
- `DesugarVisitor` Handling syntactic sugar
- `BoundCheckingVisitor` Bounds checking
- `Inline` 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/>.

 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>.



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).

Treec: An aspect-oriented approach to writing compilers.

http://dotgnu.org/treec_essay.html.

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

Names, Scopes, and Bindings

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

EPITA — École Pour l'Informatique et les Techniques Avancées

February 10, 2019

Names, Scopes, and Bindings

- 1 Bindings
- 2 Symbol Tables
- 3 Complications

Bindings

1 Bindings

- Names
- Scopes
- Binding Time

2 Symbol Tables

3 Complications

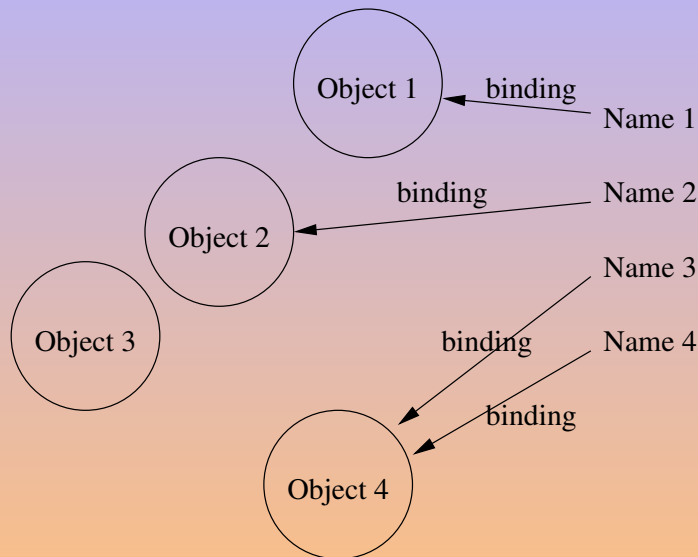
- 1 Bindings
 - Names
 - Scopes
 - Binding Time
- 2 Symbol Tables
- 3 Complications

Names, Identifiers, Symbols

- *Name (Identifiers, Symbols)*
- *reference*
- *address*
- *value*
- To refer to some entities: variable, type, function, namespace, constant, control structure (e.g., named next, continue in Perl), etc.

- usually alphanumeric and underscore, letter first, without white spaces.
- ALGOL 60, FORTRAN ignore white spaces.
- limitation on the length
 - 6 characters for the original FORTRAN (Fortran 90: 31),
 - ISO C: 31
 - no limit for most others.
- case insensitive in Modula-2 and Ada.

Names, Objects, and Bindings [Edwards, 2003]



Names, Objects, and Bindings

- When are objects created and destroyed?
Lifetimes (deferred to a later lecture).
- When are names created and destroyed?
Scopes.
- When are bindings created and destroyed?
Binding times.

Names, Objects, and Bindings

- When are objects created and destroyed?
Lifetimes (deferred to a later lecture).
- When are names created and destroyed?
Scopes.
- When are bindings created and destroyed?
Binding times.

Names, Objects, and Bindings

- When are objects created and destroyed?
Lifetimes (deferred to a later lecture).
- When are names created and destroyed?
Scopes.
- When are bindings created and destroyed?
Binding times.

Scopes

1 Bindings

- Names
- **Scopes**
- Binding Time

2 Symbol Tables

3 Complications

When are names created, visible, and destroyed?

Scope

The **textual** region in the **source** in which the binding is active.

Static Scoping

The scope can be computed at compile-time.

Dynamic Scoping

The scope depends on runtime conditions such as the function calls.

Scopes [Edwards, 2003]

When are names created, visible, and destroyed?

Scope

The **textual** region in the **source** in which the binding is active.

Static Scoping

The scope can be computed at **compile-time**.

Dynamic Scoping

The scope depends on runtime conditions such as the function calls.

Scopes [Edwards, 2003]

When are names created, visible, and destroyed?

Scope

The **textual** region in the **source** in which the binding is active.

Static Scoping

The scope can be computed at **compile-time**.

Dynamic Scoping

The scope depends on runtime conditions such as the function **calls**.

Why Scopes?

- Scopes are the first form of structure/modularity
- No scopes in assembly
- No scopes in MFS
(First generation of the Macintosh File System)
- Without scopes, names have a global influence
- With scopes, the programmer can focus on local influences
- Scopes in correct programs with unique identifiers are “useless”
- C++ namespaces are “pure scopes”

Why Scopes?

- Scopes are the first form of structure/modularity
- No scopes in assembly
- No scopes in MFS
(First generation of the Macintosh File System)
- Without scopes, names have a global influence
- With scopes, the programmer can focus on local influences
- Scopes in correct programs with unique identifiers are “useless”
- C++ namespaces are “pure scopes”

Why Scopes?

- Scopes are the first form of structure/modularity
- No scopes in assembly
- No scopes in MFS
(First generation of the Macintosh File System)
- Without scopes, names have a global influence
- With scopes, the programmer can focus on **local** influences
- Scopes in correct programs with unique identifiers are “useless”
- C++ namespaces are “pure scopes”

Why Scopes?

- Scopes are the first form of structure/modularity
- No scopes in assembly
- No scopes in MFS
(First generation of the Macintosh File System)
- Without scopes, names have a global influence
- With scopes, the programmer can focus on **local** influences
- Scopes in correct programs with unique identifiers are “useless”
- C++ namespaces are “pure scopes”

Why Scopes?

- Scopes are the first form of structure/modularity
- No scopes in assembly
- No scopes in MFS
(First generation of the Macintosh File System)
- Without scopes, names have a global influence
- With scopes, the programmer can focus on **local** influences
- Scopes in correct programs with unique identifiers are “useless”
- C++ namespaces are “pure scopes”

Why Scopes?

- Scopes are the first form of structure/modularity
- No scopes in assembly
- No scopes in MFS
(First generation of the Macintosh File System)
- Without scopes, names have a global influence
- With scopes, the programmer can focus on **local** influences
- Scopes in correct programs with unique identifiers are “useless”
- C++ namespaces are “pure scopes”

Why Scopes?

- Scopes are the first form of structure/modularity
- No scopes in assembly
- No scopes in MFS
(First generation of the Macintosh File System)
- Without scopes, names have a global influence
- With scopes, the programmer can focus on **local** influences
- Scopes in correct programs with unique identifiers are “useless”
- C++ namespaces are “pure scopes”

Declaration

Blocks determine scopes.

- local variables
- non local variables
- global variables

```
int global;

auto outer(void)
{
    int local, non_local;

    int inner(void)
    {
        return global + non_local;
    }

    return inner;
}
```

Static Scoping

- In most languages
(Ada, C, Tiger, FORTRAN, Scheme, Perl (my), etc.).
- Enables static binding.
- Enables static typing.
- Enables strong typing (Ada, ALGOL 68, Tiger).
 - safer
 - faster
 - clearer

Static Scoping

- In most languages
(Ada, C, Tiger, FORTRAN, Scheme, Perl (my), etc.).
- Enables static binding.
- Enables static typing.
- Enables strong typing (Ada, ALGOL 68, Tiger).
 - safer
 - faster
 - clearer

Static Scoping

- In most languages
(Ada, C, Tiger, FORTRAN, Scheme, Perl (my), etc.).
- Enables static binding.
- **Enables static typing.**
- Enables strong typing (Ada, ALGOL 68, Tiger).
 - safer
 - faster
 - clearer

Static Scoping

- In most languages
(Ada, C, Tiger, FORTRAN, Scheme, Perl (my), etc.).
- Enables static binding.
- Enables static typing.
- Enables strong typing (Ada, ALGOL 68, Tiger).
 - safer
 - faster
 - clearer

Static Scoping

- In most languages
(Ada, C, Tiger, FORTRAN, Scheme, Perl (my), etc.).
- Enables static binding.
- Enables static typing.
- Enables strong typing (Ada, ALGOL 68, Tiger).
 - safer
 - faster
 - clearer

Static Scoping

- In most languages
(Ada, C, Tiger, FORTRAN, Scheme, Perl (my), etc.).
- Enables static binding.
- Enables static typing.
- Enables strong typing (Ada, ALGOL 68, Tiger).
 - safer
 - faster
 - clearer

Static Scoping

- In most languages
(Ada, C, Tiger, FORTRAN, Scheme, Perl (my), etc.).
- Enables static binding.
- Enables static typing.
- Enables strong typing (Ada, ALGOL 68, Tiger).
 - safer
 - faster
 - clearer

Dynamic Scoping

- In most scripting/interpreted languages (Perl (local), Shell Script, T_EX etc.) but also in Lisp (as opposed to Scheme).

Dynamic Scoping in TeX

```
% \x, \y undefined.
{
  % \x, \y undefined.
  \def \x 1
  % \x defined, \y undefined.
  \ifnum \a < 42
    \def \y 51
  \fi
  % \x defined, \y may be defined.
}
% \x, \y undefined.
```

- Prevents static typing

An identifier may refer to different values, with different types.

Dynamic Scoping

- In most scripting/interpreted languages (Perl (local), Shell Script, T_EX etc.) but also in Lisp (as opposed to Scheme).

Dynamic Scoping in TeX

```
% \x, \y undefined.
{
  % \x, \y undefined.
  \def \x 1
  % \x defined, \y undefined.
  \ifnum \a < 42
    \def \y 51
  \fi
  % \x defined, \y may be defined.
}
% \x, \y undefined.
```

- Prevents static typing

An identifier may refer to different values, with different types.

Dynamic Scoping

- In most scripting/interpreted languages (Perl (local), Shell Script, T_EX etc.) but also in Lisp (as opposed to Scheme).

Dynamic Scoping in TeX

```
% \x, \y undefined.
{
  % \x, \y undefined.
  \def \x 1
  % \x defined, \y undefined.
  \ifnum \a < 42
    \def \y 51
  \fi
  % \x defined, \y may be defined.
}
% \x, \y undefined.
```

- Prevents static typing

An identifier may refer to different values, with different types.

Scopes in Tiger

Many different `t`, including several “variables”.

`t time`

```
let
  type      t = { h: int, t: t }
  function  t (h: int, t: t) : t =
    t { h = h, t = t }

  var      t := t (12, nil)
  var      t := t (12, t)
in
  t.t = t
end
```

Scopes [Appel, 1998]

ML

```
structure M = struct
  structure E = struct
    val a = 5;
  end
  structure N = struct
    val b = 10;
    val a = E.a + b;
  end
  structure D = struct
    val d = E.a + N.a;
  end
end
```

Java (fwd declaration allowed)

```
package M;
class E {
  static int a = 5;
}
class N {
  static int b = 10;
  static int a = E.a + b;
}
class D {
  static int d = E.a + N.a;
}
```

Scopes [Appel, 1998]

```
structure M = struct
  structure E = struct
    val a = 5;
  end
  structure N = struct
    val b = 10;
    val a = E.a + b;
  end
  structure D = struct
    val d = E.a + N.a;
  end
end
```

σ_0 = Prelude

σ_1 = $\{a : int\}$

σ_2 = $\{E : \sigma_1\}$

σ_3 = $\{b : int, a : int\}$

σ_4 = $\{N : \sigma_3\}$

σ_5 = $\{d : int\}$

σ_6 = $\{D : \sigma_5\}$

σ_7 = $\sigma_2 + \sigma_4 + \sigma_6$

$\sigma_0 + \sigma_2 \vdash N : \sigma_3$ (ML)

$\sigma_0 + \sigma_2 + \sigma_4 \vdash N : \sigma_3$ (Java)

$\sigma_0 + \sigma_2 + \sigma_4 + \sigma_6 \vdash M : \sigma_7$

Lifetime (or extent)

- Lifetime is a different matter, related to the execution (as opposed to visibility).
- Extent bound to lifetime of block tend to promote global variables (Pascal).
- *Static local variables* as in C (static), ALGOL 60 own, PL/I. Initialization?
- Modules tend to replace this block related feature.

Lifetime (or extent)

- Lifetime is a different matter, related to the execution (as opposed to visibility).
- Extent bound to lifetime of block tend to promote global variables (Pascal).
- *Static local variables* as in C (static), ALGOL 60 own, PL/I. Initialization?
- Modules tend to replace this block related feature.

Lifetime (or extent)

- Lifetime is a different matter, related to the execution (as opposed to visibility).
- Extent bound to lifetime of block tend to promote global variables (Pascal).
- *Static local variables* as in C (static), ALGOL 60 own, PL/I.
Initialization?
- Modules tend to replace this block related feature.

Lifetime (or extent)

- Lifetime is a different matter, related to the execution (as opposed to visibility).
- Extent bound to lifetime of block tend to promote global variables (Pascal).
- *Static local variables* as in C (static), ALGOL 60 own, PL/I. Initialization?
- Modules tend to replace this block related feature.

Lifetime (or extent)

- Lifetime is a different matter, related to the execution (as opposed to visibility).
- Extent bound to lifetime of block tend to promote global variables (Pascal).
- *Static local variables* as in C (static), ALGOL 60 own, PL/I. Initialization?
- Modules tend to replace this block related feature.

Scopes in C++

A lot of scope in a real programming language.

- Block scope
- Function parameter scope
- Function scope
- Namespace scope
- Class scope
- Enumeration Scope
- Template parameter scope

Scopes in C++

A lot of scope in a real programming language.

- Block scope
- Function parameter scope
- Function scope
- Namespace scope
- Class scope
- Enumeration Scope
- Template parameter scope

Scopes in C++

A lot of scope in a real programming language.

- Block scope
- Function parameter scope
- Function scope
- Namespace scope
- Class scope
- Enumeration Scope
- Template parameter scope

Scopes in C++

A lot of scope in a real programming language.

- Block scope
- Function parameter scope
- Function scope
- Namespace scope
- Class scope
- Enumeration Scope
- Template parameter scope

Scopes in C++

A lot of scope in a real programming language.

- Block scope
- Function parameter scope
- Function scope
- Namespace scope
- Class scope
- Enumeration Scope
- Template parameter scope

Scopes in C++

A lot of scope in a real programming language.

- Block scope
- Function parameter scope
- Function scope
- Namespace scope
- Class scope
- Enumeration Scope
- Template parameter scope

Scopes in C++

A lot of scope in a real programming language.

- Block scope
- Function parameter scope
- Function scope
- Namespace scope
- Class scope
- Enumeration Scope
- Template parameter scope

Scopes in C++

A lot of scope in a real programming language.

- Block scope
- Function parameter scope
- Function scope
- Namespace scope
- Class scope
- Enumeration Scope
- Template parameter scope

Binding Time

1 Bindings

- Names
- Scopes
- Binding Time

2 Symbol Tables

3 Complications

Binding Time [Edwards, 2003]

When a binding from a name to an object is made.

Binding Time	Examples
language design	if
language implementation	data width
program writing	foo, bar
compilation	static objects, code
linkage	relative addresses
loading	shared objects
execution	heap objects

Binding Time: the moving IN

Roughly, flexibility and efficiency

- are mutually exclusive
- depend on binding time.

The Moving IN

binding-time

early -----> late

INflexibility flexibility
efficiency INefficiency

Binding Time: the moving IN

Roughly, flexibility and efficiency

- are mutually exclusive
- depend on binding time.

The Moving IN

binding-time

early -----> late

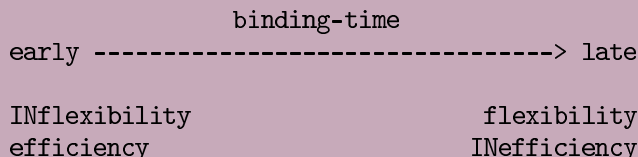
INflexibility flexibility
efficiency INefficiency

Binding Time: the moving IN

Roughly, flexibility and efficiency

- are mutually exclusive
- depend on binding time.

The Moving IN



Dynamic Binding: virtual in C++

Dynamic dispatch is roughly **runtime** overloading.

Dynamic Dispatch in C++

```
struct Shape
{
    virtual void draw() const = 0;
};
```

```
struct Square : public Shape
{
    void draw() const override {};
};
```

```
struct Circle : public Shape
{
    void draw() const override {};
};
```

Dynamic Binding: virtual in C++

Dynamic Dispatch in C++

```
#include <vector>
#include "shapes.hh"

using shapes_type = std::vector<Shape*>;

int main()
{
    auto ss = shapes_type{new Circle, new Square};

    for (auto s: ss)
        // Inclusion polymorphism.
        s->draw();
}
```


Late Code Binding: eval

- Most interpreted languages support eval (explicit or not): runtime code evaluation.
- Enables language extensions.

try/catch in Perl

```
try {  
    die "phooey";  
} catch {  
    /phooey/ and print "unphooey\n";  
};
```

Late Code Binding: eval

- Most interpreted languages support eval (explicit or not): runtime code evaluation.
- Enables language extensions.

try/catch in Perl

```
sub try ($@) {
    my ($try, $catch) = @_;
    eval { $try };
    if ($?) {
        local $_ = $@;
        $catch;
    }
}

sub catch ($@) {
    $!0;
}

try {
    die "phooey";
} catch {
    /phooey/ and print "unphooey\n";
};
```

- Most interpreted languages support eval (explicit or not): runtime code evaluation.
- Enables language extensions.

try/catch in Perl

```
sub try (&@) {  
    my ($try, $catch) = @_;  
    eval { &$try }; # Explicit eval.  
    if ($?) {  
        local $_ = $@;  
        &$catch;  
    }  
}  
  
sub catch (&) {  
    $_[0]; # implicit eval.  
}  
  
try {  
    die "phooey";  
} catch {  
    /phooey/ and print "unphooey\n";  
};
```

- Most interpreted languages support eval (explicit or not): runtime code evaluation.
- Enables language extensions.

try/catch in Perl

```
sub try (&@) {  
    my ($try, $catch) = @_;  
    eval { &$try }; # Explicit eval.  
    if ($?) {  
        local $_ = $?;  
        &$catch;  
    }  
}  
  
sub catch (&) {  
    $_[0]; # implicit eval.  
}  
  
try {  
    die "phooey";  
} catch {  
    /phooey/ and print "unphooey\n";  
};
```

Binding Times in Tiger [Edwards, 2003]

Design Keywords

Program Identifiers

Compile Function code, frames, types

Execution Records, arrays addresses

Little dynamic behavior

Symbol Tables

- 1 Bindings
- 2 Symbol Tables
- 3 Complications

Visiting an ast

For statically scoped languages

- many traversals check **uses** against **definitions**
- most traversals need a form of memory (binding, type, escapes, inlining, translation, etc.)
- this memory is related to scopes
- we need some reversible memory (do/undo)

Similarly for narrow compilers without ast

Visiting an ast

For statically scoped languages

- many traversals check **uses** against **definitions**
- most traversals need a form of memory (binding, type, escapes, inlining, translation, etc.)
- this memory is related to scopes
- we need some reversible memory (do/undo)

Similarly for narrow compilers without ast

Visiting an ast

For statically scoped languages

- many traversals check **uses** against **definitions**
- most traversals need a form of memory (binding, type, escapes, inlining, translation, etc.)
- this memory is related to scopes
 - we need some reversible memory (do/undo)

Similarly for narrow compilers without ast

Visiting an ast

For statically scoped languages

- many traversals check **uses** against **definitions**
- most traversals need a form of memory (binding, type, escapes, inlining, translation, etc.)
- this memory is related to scopes
- we need some reversible memory (do/undo)

Similarly for narrow compilers without ast

Visiting an ast

For statically scoped languages

- many traversals check **uses** against **definitions**
- most traversals need a form of memory (binding, type, escapes, inlining, translation, etc.)
- this memory is related to scopes
- we need some reversible memory (do/undo)

Similarly for narrow compilers without ast

Symbol Tables: Scopes

Handle scopes?

- not needed if all the names are unique
- or if there exists a unique identifier
- required otherwise

Handle scopes explicitly?

- yes: the tables support undecorated symbol tables
- no: rely on automatic variables

Symbol Tables: Scopes

Handle scopes?

- not needed if all the names are unique
- or if there exists a unique **identifier**
- required otherwise

Handle scopes explicitly?

- yes: the tables support undecorated symbol tables
- no: rely on automatic variables

Symbol Tables: Scopes

Handle scopes?

- not needed if all the names are unique
- or if there exists a unique identifier
- required otherwise

Handle scopes explicitly?

- yes: the tables support undo: scoped symbol tables
- no: rely on automatic variables

Symbol Tables: Scopes

Handle scopes?

- not needed if all the names are unique
- or if there exists a unique identifier
- required otherwise

Handle scopes explicitly?

- yes: the tables support undo: scoped symbol tables
- no: rely on automatic variables

Symbol Tables: Scopes

Handle scopes?

- not needed if all the names are unique
- or if there exists a unique identifier
- required otherwise

Handle scopes explicitly?

- **yes:** the tables support undo:
scoped symbol tables
- **no:** rely on automatic variables

Symbol Tables: Scopes

Handle scopes?

- not needed if all the names are unique
- or if there exists a unique identifier
- required otherwise

Handle scopes explicitly?

- **yes:** the tables support undo: scoped symbol tables
- **no:** rely on automatic variables

(Non Scoped) Symbol Tables

An associative array

- put
- get

Implementation

- a list
- a tree
- a hash
- ...

(Non Scoped) Symbol Tables

An associative array

- put
- get

Implementation

- a list
- a tree
- a hash
- ...

(Non Scoped) Symbol Tables

An associative array

- put
- get

Implementation

- a list
- a tree
- a hash
- ...

(Non Scoped) Symbol Tables

An **associative array**

- put
- get

Implementation

- a list
- a tree
- a hash
- ...

(Non Scoped) Symbol Tables

An associative array

- put
- get

Implementation

- a list
- a tree
- a hash
- ...

(Non Scoped) Symbol Tables

An associative array

- put
- get

Implementation

- a list
- a tree
- a hash
- ...

(Non Scoped) Symbol Tables

An associative array

- put
- get

Implementation

- a list
- a tree
- a hash
- ...

(Non Scoped) Symbol Tables

An associative array

- put
- get

Implementation

- a list
- a tree
- a hash
- ...

Scoped Symbol Table: `symbol::Table`

class Table

```
template <typename Entry_T>
class Table
{
public:
    Table();

    auto put(symbol key, Entry_T& val) -> void;
    auto get(symbol key) const -> Entry_T*;

    auto scope_begin() -> void;
    auto scope_end() -> void;

    auto print(std::ostream& ostr) const -> void;
};
```

Not very C++ (iterators instead of pointers, `operator[]`, etc.)

Scoped Symbol Table Implementations

- Mixing Stacks and Associative Arrays
- Copying, or not copying?
- Functional (Non Destructive) Versions
- Mongrels

Scoped Symbol Table Implementations

- Mixing Stacks and Associative Arrays
- Copying, or not copying?
- Functional (Non Destructive) Versions
- Mongrels

Scoped Symbol Table Implementations

- Mixing Stacks and Associative Arrays
- Copying, or not copying?
- Functional (Non Destructive) Versions
- Mongrels

Scoped Symbol Table Implementations

- Mixing Stacks and Associative Arrays
- Copying, or not copying?
- Functional (Non Destructive) Versions
- Mongrels

Memory Management

When do you deallocate associated data?

`scope end` deallocate everything since the latest `scope_begin`

`pass end` deallocate auxiliary data after the traversal is completed

`ast` bind the data to the `ast` and delegate deallocation

by hand thanks God for Valgrind

never

Memory Management

When do you deallocate associated data?

`scope end` deallocate everything since the latest `scope_begin`

`pass end` deallocate auxiliary data after the traversal is completed

`ast` bind the data to the `ast` and delegate deallocation

by hand thanks God for Valgrind

never

When do you deallocate associated data?

`scope end` deallocate everything since the latest `scope_begin`

`pass end` deallocate auxiliary data after the traversal is completed

`ast` bind the data to the ast and delegate deallocation

`by hand` thanks God for Valgrind

`never`

Memory Management

When do you deallocate associated data?

`scope end` deallocate everything since the latest `scope_begin`

`pass end` deallocate auxiliary data after the traversal is completed

`ast` bind the data to the ast and delegate deallocation

`by hand` thanks God for Valgrind

never

When do you deallocate associated data?

`scope end` deallocate everything since the latest `scope_begin`

`pass end` deallocate auxiliary data after the traversal is completed

`ast` bind the data to the ast and delegate deallocation

`by hand` thanks God for Valgrind and Paracetamol

never

When do you deallocate associated data?

`scope end` deallocate everything since the latest `scope_begin`

`pass end` deallocate auxiliary data after the traversal is completed

`ast` bind the data to the ast and delegate deallocation

`by hand` thanks God for Valgrind and Paracetamol

`never`

When do you deallocate associated data?

- `scope end` deallocate everything since the latest `scope_begin`

- `pass end` deallocate auxiliary data after the traversal is completed

- `ast` bind the data to the ast and delegate deallocation

- `by hand` thanks God for Valgrind and Paracetamol

- `never` tu sors

Memory Management: Deallocate on scope exit

But then...

Twice foo

```
let var foo := 42
    var foo := 51
in foo end
```

Two lets

```
let var foo := 42 in
let var foo := 51
in foo end end
```

but then again...

Escaping type

```
let type rec = {}
in rec {} end <> nil
```

Segmentation violation...

Memory Management: Deallocate on scope exit

But then...

Twice foo

```
let var foo := 42  
    var foo := 51  
in foo end
```

Two lets

```
let var foo := 42 in  
let var foo := 51  
in foo end end
```

but then again...

Escaping type

```
let type rec = {}  
in rec {} end <> nil
```

Segmentation violation...

Memory Management: Deallocate on scope exit

But then...

Twice foo

```
let var foo := 42
    var foo := 51
in foo end
```

Two lets

```
let var foo := 42 in
let var foo := 51
in foo end end
```

but then again...

Escaping type

```
let type rec = {}
in rec {} end <> nil
```

Segmentation violation...

Memory Management: Deallocate on scope exit

But then...

Twice foo

```
let var foo := 42
    var foo := 51
in foo end
```

Two lets

```
let var foo := 42 in
let var foo := 51
in foo end end
```

but then again...

Escaping type

```
let type rec = {}
in rec {} end <> nil
```

Segmentation violation...
Courtesy of Arnaud Fabre.

Memory Management: Deallocate with the AST

- annotate each node of ast
- annotate each scoping node with a symbol table and link them
- leave tables outside

Memory Management: Deallocate with the AST

- annotate each node of ast
- annotate each scoping node with a symbol table and link them
- leave tables outside

Memory Management: Deallocate with the AST

- annotate each node of ast
- annotate each scoping node with a symbol table and link them
- leave tables outside

Factoring Scope Handling

- no scope handling needed if names are unique
- so use regular associative containers
- but how can you guarantee unique names
- do you need to make names uniques?

Factoring Scope Handling

- no scope handling needed if names are unique
- so use **regular** associative containers
- but how can you guarantee unique names
- do you need to make names uniques?

Factoring Scope Handling

- no scope handling needed if names are unique
- so use regular associative containers
- but how can you guarantee unique names
- do you need to make names uniques?

Factoring Scope Handling

- no scope handling needed if names are unique
- so use regular associative containers
- but how can you guarantee unique names
- do you need to make names uniques?

Factoring Scope Handling

- no scope handling needed if names are unique
- so use regular associative containers
- but how can you guarantee unique names
- do you need to make names uniques?

Bind the names/Label by definition address

- annotates uses with links to their definitions
- uses scoped symbol tables
- or regular containers and recursion
- checks multiple definitions
- checks missing definitions
- and also binds...

- annotates uses with links to their definitions
- uses scoped symbol tables
 - or regular containers and recursion
- checks multiple definitions
- checks missing definitions
- and also binds...

- annotates uses with links to their definitions
- uses scoped symbol tables
- or regular containers **and** recursion
- checks multiple definitions
- checks missing definitions
- and also binds...

- annotates uses with links to their definitions
- uses scoped symbol tables
- or regular containers **and** recursion
- checks multiple definitions
- checks missing definitions
- and also binds...

- annotates uses with links to their definitions
- uses scoped symbol tables
- or regular containers **and** recursion
- checks multiple definitions
- checks missing definitions
- and also binds...

- annotates uses with links to their definitions
- uses scoped symbol tables
- or regular containers **and** recursion
- checks multiple definitions
- checks missing definitions
- and also binds...

- annotates uses with links to their definitions
- uses scoped symbol tables
- or regular containers **and** recursion
- checks multiple definitions
- checks missing definitions
- and also binds... breaks to their loops

Complications

1 Bindings

2 Symbol Tables

3 **Complications**

- Overloading
- Non Local Variables

Overloading

1 Bindings

2 Symbol Tables

3 **Complications**

- **Overloading**
- Non Local Variables

Overloading

Overloading: Homonyms

Several entities bearing the same name, but **statically** distinguishable, e.g., by their arity, type etc.

```
// foo is overloaded.  
int foo(int);  
int foo(float);
```

Aliasing: Synonyms

One entity bearing several names.

```
// x and y are aliases.  
int x;  
int& y = x;
```

Operator Overloading

Overloading is meant to simplify the user's life. Since FORTRAN!

Overloading in Caml

```
# 1 + 2;;  
- : int = 3  
# 1.0 + 2.0;;  
Characters 0-3:  
  1.0 + 2.0;;  
  ^^^
```

This expression has type float but is here used with type int

```
# 1.0 +. 2.0;;  
- : float = 3.
```

Thank God, C was invented to improve Caml:

```
int    a = 1    + 2;;  
float b = 1.0 + 2.0;;
```

Operator Overloading

Overloading is meant to simplify the user's life. Since FORTRAN!

Overloading in Caml

```
# 1 + 2;;
```

```
- : int = 3
```

```
# 1.0 + 2.0;;
```

```
Characters 0-3:
```

```
  1.0 + 2.0;;
```

```
  ^^^
```

This expression has type float but is here used with type int

```
# 1.0 +. 2.0;;
```

```
- : float = 3.
```

Thank God, C was invented to improve Caml:

```
int    a = 1    + 2;;
```

```
float b = 1.0 + 2.0;;
```

Operator Overloading

Overloading is meant to simplify the user's life. Since FORTRAN!

Overloading in Caml

```
# 1 + 2;;  
- : int = 3  
# 1.0 + 2.0;;  
Characters 0-3:  
  1.0 + 2.0;;  
  ^^^
```

This expression has type float but is here used with type int

```
# 1.0 +. 2.0;;  
- : float = 3.
```

Thank God, C was invented to improve Caml:

```
int    a = 1    + 2;;  
float b = 1.0 + 2.0;;
```

Of course this is unfair: Caml has type inference.

Function Overloading

Usually based on the arguments
(Ada, C++, Java...; not C, ALGOL 60, Fortran...).

ALGOL 60

```
integer I;  
real X;  
...  
PUTSTRING("results are: ");  PUTINT(I);  PUTREAL(X);
```

Ada [ARM, 1983]

```
I : INTEGER;  
X : REAL;  
...  
PUT("results are: ");  PUT(I);  PUT(X);
```

Overloading is Syntactic Sugar

Overloaded

```
#include <string>

void foo(int);
void foo(char);
void foo(const char*);
void foo(std::string);

int
main ()
{
    foo(0);
    foo('0');
    foo("0");
    foo(std::string("0"));
}
```

Un-overloaded

```
#include <string>

void foo_int(int);
void foo_char(char);
void foo_char_p(const char*);
void foo_std_string(std::string);

int
main ()
{
    foo_int(0);
    foo_char('0');
    foo_char_p("0");
    foo_std_string(std::string("0"));
}
```


Overloading is Syntactic Sugar

Overloaded

```
#include <string>

void foo(int);
void foo(char);
void foo(const char*);
void foo(std::string);

int
main ()
{
    foo(0);
    foo('0');
    foo("0");
    foo(std::string("0"));
}
```

Un-overloaded

```
#include <string>

void foo_int(int);
void foo_char(char);
void foo_char_p(const char*);
void foo_std_string(std::string);

int
main ()
{
    foo_int(0);
    foo_char('0');
    foo_char_p("0");
    foo_std_string(std::string("0"));
}
```

Overloading is Syntactic Sugar

Usually solved by renaming/mangling.

g++-2.95, como

```
f__Fi  -> int f(int);  
f__FPc -> int f(char*);
```

g++-3.2, icc

```
_Z1fi  -> int f(int);  
_Z1fPc -> int f(char*);
```

Overloading in Tiger

Ordering $<$, $<=$, $>$, and $>=$

overloaded for pairs of integers, or strings.

Identity $=$ and $<>$

overloaded for (type coherent) pairs of integers, strings, arrays or records.

Non Local Variables

1 Bindings

2 Symbol Tables

3 Complications

- Overloading
- Non Local Variables

Lambda Shifting

With nested functions

```
int global;

int outer(void)
{
    int local, non_local;

    int inner(void)
    {
        return
            global + non_local;
    }

    return inner();
}
```

Without

```
int global;

int outer_inner_(int* non_local)
{
    return global + *non_local;
}

int outer(void)
{
    int local, non_local;
    return outer_inner_(&non_local);
}
```

Lambda Shifting

With nested functions

```
int global;

int outer(void)
{
    int local, non_local;

    int inner(void)
    {
        return
            global + non_local;
    }

    return inner();
}
```

Without

```
int global;

int outer_inner_(int* non_local)
{
    return global + *non_local;
}

int outer(void)
{
    int local, non_local;
    return outer_inner_(&non_local);
}
```

Non Local Variables

```
let
  function outer(): int =
    let
      nonlocal var outer := 0
    in
      let
        function inner() : int =
          let
            var inner := 1
          in
            inner + outer
          end
        in
          inner()
        end
      end
    in
      outer ()
  end
```

Non Non Local Variables

```
let
```

```
  let
```

```
    local var outer := 0
```

```
  in
```

```
    let
```

```
      let
```

```
        var inner := 1
```

```
      in
```

```
        inner + outer
```

```
      end
```

```
    in
```

```
  end
```

```
end
```

```
in
```

```
end
```


Non Non Local Variables

```
let
  function outer(): int =
    let
      local var outer := 0
    in
      let
        let
          var inner := 1
        in
          inner + outer
        end
      in
        end
      end
    in
      outer()
  end
```

The Escapes and Functional Programming

```
let
  function add(nonlocal a: int, b: int) : int =
    let
      function add_a(x: int) : int = a + x
    in
      add_a(b)
    end
in
  print_int(add(1, 2));
  print("\n")
end
```

Closures

```
let
  function add_gen(nonlocal a: int) : int -> int =
    let
      function add_a(x: int) : int = a + x
    in
      add_a
    end
  incr = add_gen(1);
in
  print_int(incr(2));
  print("\n");
end
```

The Escapes & Recursion

```
let
  function one(input : int) =
    let
      function two() =
        (print("two: "); print_int(input);
         print("\n");
         one(input))
    in
      if input > 0 then
        (input := input - 1;
         two(); print("one: ");
         print_int(input); print("\n"))
      end
    in
      one (3)
  end
```

Escaping Variables/Arguments

Technically **escaping** means “cannot be stored in a register”.

In C

- Large values (arrays, structs).
- Variables whose address is taken.
- Variable arguments.

In Tiger

- variables/arguments from outer functions.
- not variables/arguments from outer scopes.

Escaping Variables/Arguments

Technically **escaping** means “cannot be stored in a register”.

In C

- Large values (arrays, structs).
- Variables whose address is taken.
- Variable arguments.

In Tiger

- variables/arguments from outer functions.
- not variables/arguments from outer scopes.

Escaping Variables/Arguments

Technically **escaping** means “cannot be stored in a register”.

In C

- Large values (arrays, structs).
- Variables whose address is taken.
- Variable arguments.

In Tiger

- variables/arguments from outer functions.
- not variables/arguments from outer scopes.

Escaping Variables/Arguments

Technically **escaping** means “cannot be stored in a register”.

In C

- Large values (arrays, structs).
- Variables whose address is taken.
- Variable arguments.

In Tiger

- variables/arguments from outer functions.
- not variables/arguments from outer scopes.

Escaping Variables/Arguments

Technically **escaping** means “cannot be stored in a register”.

In C

- Large values (arrays, structs).
- Variables whose address is taken.
- Variable arguments.

In Tiger

- variables/arguments from outer **functions**.
- not variables/arguments from outer **scopes**.

Escaping Variables/Arguments

Technically **escaping** means “cannot be stored in a register”.

In C

- Large values (arrays, structs).
- Variables whose address is taken.
- Variable arguments.

In Tiger

- variables/arguments from outer **functions**.
- not variables/arguments from outer **scopes**.

Escaping Variables/Arguments

Technically **escaping** means “cannot be stored in a register”.

In C

- Large values (arrays, structs).
- Variables whose address is taken.
- Variable arguments.

In Tiger

- variables/arguments from outer **functions**.
- **not** variables/arguments from outer **scopes**.

Annotating the ast

- being non local means having non local uses
- obviously non local variables need to be accessible from inner functions
- to simplify the compiler, it is easier to leave them on the stack
- hence the translation to intermediate representation needs to know which variables are non local from their definitions
- therefore a preliminary pass should flag non local variables

Annotating the ast

- being non local means having non local uses
- obviously non local variables need to be accessible from inner functions
- to simplify the compiler, it is easier to leave them on the stack
- hence the translation to intermediate representation needs to know which variables are non local from their definitions
- therefore a preliminary pass should flag non local variables

Annotating the ast

- being non local means having non local uses
- obviously non local variables need to be accessible from inner functions
- to simplify the compiler, it is easier to leave them on the stack
- hence the translation to intermediate representation needs to know which variables are non local from their definitions
- therefore a preliminary pass should flag non local variables

Annotating the ast

- being non local means having non local uses
- obviously non local variables need to be accessible from inner functions
- to simplify the compiler, it is easier to leave them on the stack
- hence the translation to intermediate representation needs to know which variables are non local from their definitions
- therefore a preliminary pass should flag non local variables

Annotating the ast

- being non local means having non local **uses**
- obviously non local variables need to be accessible from inner functions
- to simplify the compiler, it is easier to leave them on the stack
- hence the translation to intermediate representation needs to know which variables are non local from their **definitions**
- therefore a preliminary pass should flag non local variables



Appel, A. W. (1998).

Modern Compiler Implementation in C, Java, ML.

Cambridge University Press.



ARM (1983).

Ada Reference Manual.



Edwards, S. (2003).

COMS W4115 Programming Languages and Translators.

<http://www.cs.columbia.edu/~sedwards/classes/2003/w4115/>.

Types

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

EPITA — École Pour l'Informatique et les Techniques Avancées

January 23, 2019

- 1 Why using Types?
- 2 What is Type Checking?
- 3 Type Inference (Crash Introduction to Natural Deduction)
- 4 Type Checking in Practice
- 5 An Overview of Types

Why using Types?

- 1 Why using Types?
- 2 What is Type Checking?
- 3 Type Inference (Crash Introduction to Natural Deduction)
- 4 Type Checking in Practice
- 5 An Overview of Types

Types are not necessary!

Consider the assembly language fragment:

addi \$r1, \$r2, \$r3

What are the types of \$r1, \$r2, \$r3?

Assembly language is untyped (MIPS assembly) !

Types are not necessary!

Consider the assembly language fragment:

```
addi $r1, $r2, $r3
```

What are the types of \$r1, \$r2, \$r3?

Assembly language is untyped (MIPS assembly) !

Why do we need type systems?

- It does not make sense to add a function pointer and an integer in C
- It does make sense to add two integers
- But both have the same assembly language implementation!

Why do we need type systems?

- It does not make sense to add a function pointer and an integer in C
- It does make sense to add two integers
- But both have the same assembly language implementation!

Why do we need type systems?

- It does not make sense to add a function pointer and an integer in C
- It does make sense to add two integers
- But both have the same assembly language implementation!

Of FORTRAN and Satellites

ANTENNA.F

```
...  
...  
DO 1 I = 1, 5  
  
C Extend the antenna.  
...  
1  If antenna is extended  
2  Then Go to 3  
...  
3  CONTINUE  
4  ...
```

antenna.c

```
int I;  
...  
for (I = 1; I <= 5; ++I)  
{  
    /* Extend the antenna. */  
    ...  
    if (antenna_is_extended)  
        goto 3  
    ...  
3:  
    ...  
}
```

Of FORTRAN and Satellites

ANTENNA.F

```
...  
...  
DO 1 I = 1, 5  
  
C Extend the antenna.  
...  
1  If antenna is extended  
2  Then Go to 3  
...  
3  CONTINUE  
4  ...
```

antenna.c

```
int I;  
...  
for (I = 1; I <= 5; ++I)  
{  
    /* Extend the antenna. */  
    ...  
    if (antenna_is_extended)  
        goto 3  
    ...  
3:  
    ...  
}
```

Of FORTRAN and Satellites

ANTENNA.F

```
...  
...  
DO 1 I = 1. 5  
  
C Extend the antenna.  
...  
1 If antenna is extended  
2 Then Go to 3  
...  
3 CONTINUE  
4 ...
```

antenna.c

```
float D01I = 1.5;  
  
/* Extend the antenna. */  
...  
if (antenna_is_extended)  
    goto 3  
...  
3: ...
```

Russel's Paradox

$$E = \{x \notin x\} \quad E \in E \quad E \notin E$$

Based on the conjunction of:

- Any predicate is an object
- Any predicate can be applied to any object

Rejecting one leads to:

- Type theory (1909)
- Zermelo Fraenkel's set theory (1922)

Russel's Paradox

$$E = \{x \notin x\} \quad E \in E \quad E \notin E$$

Based on the conjunction of:

- Any predicate is an object
- Any predicate can be applied to any object

Rejecting one leads to:

- Type theory (1909)
- Zermelo Fraenkel's set theory (1922)

Russel's Paradox

$$E = \{x \notin x\} \quad E \in E \quad E \notin E$$

Based on the conjunction of:

- Any predicate is an object
- Any predicate can be applied to any object

Rejecting one leads to:

- Type theory (1909)
- Zermelo Fraenkel's set theory (1922)

Russel's Paradox

$$E = \{x \notin x\} \quad E \in E \quad E \notin E$$

Based on the conjunction of:

- Any predicate is an object
- Any predicate can be applied to any object

Rejecting one leads to:

- Type theory (1909)
- Zermelo Fraenkel's set theory (1922)

Russel's Paradox

$$E = \{x \notin x\} \quad E \in E \quad E \notin E$$

Based on the conjunction of:

- Any predicate is an object
- Any predicate can be applied to any object

Rejecting one leads to:

- Type theory (1909)
- Zermelo Fraenkel's set theory (1922)

Russel's Paradox

$$E = \{x \notin x\} \quad E \in E \quad E \notin E$$

Based on the conjunction of:

- Any predicate is an object
- Any predicate can be applied to any object

Rejecting one leads to:

- Type theory (1909)
- Zermelo Fraenkel's set theory (1922)

Russel's Paradox

$$E = \{x \notin x\} \quad E \in E \quad E \notin E$$

Based on the conjunction of:

- Any predicate is an object
- Any predicate can be applied to any object

Rejecting one leads to:

- Type theory (1909)
- Zermelo Fraenkel's set theory (1922)

Reject Impossible Values

- 1 1
- $\omega = \lambda x.xx$

Enable Optimizations

- Static types enables static bindings.
- E.g., + in C requires no runtime checks.

- Types are not necessary:
 - There is none at machine/assembly level
operators are “typed” though
 - There are type-less languages
e.g., in Tcl or M4 everything is a string
- But they are useful:
 - More control from the compiler
 - Catching “impossible but expressible” situations
 - Optimizing
 - Abstraction (arrays, records, etc.)
 - Memory management (automatic or not)
 - Violations of abstraction boundaries, such as using a private field from outside a class

- Types are not necessary:
 - There is none at machine/assembly level
operators are “typed” though
 - There are type-less languages
e.g., in Tcl or M4 everything is a string
- But they are useful:
 - More control from the compiler
 - Catching “impossible but expressible” situations
 - Optimizing
 - Abstraction (arrays, records, etc.)
 - Memory management (automatic or not)
 - Violations of abstraction boundaries, such as using a private field from outside a class

What is Type Checking?

- 1 Why using Types?
- 2 What is Type Checking?
- 3 Type Inference (Crash Introduction to Natural Deduction)
- 4 Type Checking in Practice
- 5 An Overview of Types

The data types of a language are a large part of what determines that language's style and usefulness (along with control structures).

- Numerics
- Booleans
- User-defined enumerations
- Subranges
- Arrays (static, stack dynamic, heap dynamic)
- Unions (discriminated/free)
- Structures/Records/Objects
- Tuples/Lists
- References/Pointers
- etc.

The data types of a language are a large part of what determines that language's style and usefulness (along with control structures).

- Numerics
- Booleans
- User-defined enumerations
- Subranges
- Arrays (static, stack dynamic, heap dynamic)
- Unions (discrimated/free)
- Structures/Records/Objects
- Tuples/Lists
- References/Pointers
- etc.

The data types of a language are a large part of what determines that language's style and usefulness (along with control structures).

- Numerics
- Booleans
- User-defined enumerations
- Subranges
- Arrays (static, stack dynamic, heap dynamic)
- Unions (discrimated/free)
- Structures/Records/Objects
- Tuples/Lists
- References/Pointers
- etc.

The data types of a language are a large part of what determines that language's style and usefulness (along with control structures).

- Numerics
- Booleans
- User-defined enumerations
- Subranges
- Arrays (static, stack dynamic, heap dynamic)
- Unions (discriminated/free)
- Structures/Records/Objects
- Tuples/Lists
- References/Pointers
- etc.

The data types of a language are a large part of what determines that language's style and usefulness (along with control structures).

- Numerics
- Booleans
- User-defined enumerations
- Subranges
- Arrays (static, stack dynamic, heap dynamic)
- Unions (discriminated/free)
- Structures/Records/Objects
- Tuples/Lists
- References/Pointers
- etc.

The data types of a language are a large part of what determines that language's style and usefulness (along with control structures).

- Numerics
- Booleans
- User-defined enumerations
- Subranges
- Arrays (static, stack dynamic, heap dynamic)
- Unions (discriminated/free)
- Structures/Records/Objects
- Tuples/Lists
- References/Pointers
- etc.

The data types of a language are a large part of what determines that language's style and usefulness (along with control structures).

- Numerics
- Booleans
- User-defined enumerations
- Subranges
- Arrays (static, stack dynamic, heap dynamic)
- Unions (discriminated/free)
- Structures/Records/Objects
- Tuples/Lists
- References/Pointers
- etc.

The data types of a language are a large part of what determines that language's style and usefulness (along with control structures).

- Numerics
- Booleans
- User-defined enumerations
- Subranges
- Arrays (static, stack dynamic, heap dynamic)
- Unions (discrimated/free)
- Structures/Records/Objects
- Tuples/Lists
- References/Pointers
- etc.

The data types of a language are a large part of what determines that language's style and usefulness (along with control structures).

- Numerics
- Booleans
- User-defined enumerations
- Subranges
- Arrays (static, stack dynamic, heap dynamic)
- Unions (discriminated/free)
- Structures/Records/Objects
- Tuples/Lists
- References/Pointers
- etc.

The data types of a language are a large part of what determines that language's style and usefulness (along with control structures).

- Numerics
- Booleans
- User-defined enumerations
- Subranges
- Arrays (static, stack dynamic, heap dynamic)
- Unions (discrimated/free)
- Structures/Records/Objects
- Tuples/Lists
- References/Pointers
- etc.

The data types of a language are a large part of what determines that language's style and usefulness (along with control structures).

- Numerics
- Booleans
- User-defined enumerations
- Subranges
- Arrays (static, stack dynamic, heap dynamic)
- Unions (discriminated/free)
- Structures/Records/Objects
- Tuples/Lists
- References/Pointers
- etc.

Type Checking

Type Checking is the activity of ensuring that the operands of an **operator** are of compatible types

A **compatible type** is one that is

- either legal for the operator
- or allowed under language rules to be implicitly converted by compiler-generated code (or the interpreter) to a legal type

Type Checking

Type Checking is the activity of ensuring that the operands of an **operator** are of compatible types

A **compatible type** is one that is

- either legal for the operator
- or allowed under language rules to be implicitly converted by compiler-generated code (or the interpreter) to a legal type

Coercion is the automatic (implicit) conversion from a type to another.

There are 2 kind of coercion in C-like languages:

- **widening** conversions: from a "smaller" type to a "larger one"

```
int i = 42;  
float f = i;
```

- **narrowing** conversions: from a "larger" type to a "smaller one"

```
float f = 42.0;  
int i = f;
```

Note: Java only allows assignment type widening coertions.

Strong Typing

A programming language is **strongly typed** if type errors are always detected.

- the types of **all** operands can be determined, either at **compile time** or at **runtime**
- detection, at run time, of uses of the incorrect type values in variables that can store values of more than one type
- Ada is nearly strongly typed due to *Unchecked_Conversion*
- C and C++ are not strongly typed languages because both include union types
- F# and ML are strongly typed

Strong Typing

A programming language is **strongly typed** if type errors are always detected.

- the types of **all** operands can be determined, either at **compile time** or at **runtime**
- detection, at run time, of uses of the incorrect type values in variables that can store values of more than one type
- Ada is nearly strongly typed due to *Unchecked_Conversion*
- C and C++ are not strongly typed languages because both include union types
- F# and ML are strongly typed

Strong Typing

A programming language is **strongly typed** if type errors are always detected.

- the types of **all** operands can be determined, either at **compile time** or at **runtime**
- detection, at run time, of uses of the incorrect type values in variables that can store values of more than one type
- Ada is nearly strongly typed due to *Unchecked_Conversion*
- C and C++ are not strongly typed languages because both include union types
- F# and ML are strongly typed

Strong Typing

A programming language is **strongly typed** if type errors are always detected.

- the types of **all** operands can be determined, either at **compile time** or at **runtime**
- detection, at run time, of uses of the incorrect type values in variables that can store values of more than one type
- Ada is nearly strongly typed due to *Unchecked_Conversion*
- C and C++ are not strongly typed languages because both include union types
- F# and ML are strongly typed

Strong Typing

A programming language is **strongly typed** if type errors are always detected.

- the types of **all** operands can be determined, either at **compile time** or at **runtime**
- detection, at run time, of uses of the incorrect type values in variables that can store values of more than one type
- Ada is nearly strongly typed due to *Unchecked_Conversion*
- C and C++ are not strongly typed languages because both include union types
- F# and ML are strongly typed

Type Equivalence

Two types are **equivalent** if an operand of one type in an expression is substituted for one of the other type, without coercion.

In other words, **Type equivalence** is a strict form of compatibility type compatibility without coercion.

- **Name type equivalence**

- two variables have equivalent types if they are defined either in the same declaration or in declarations that use the same type name

```
int i = 42; int j = 51;
```

- **Structure type equivalence**

- two variables have equivalent types if their types have identical structures

```
using celcius = int;  
int i = 42; celcius j = 51;
```

Static Typing vs. Dynamic Typing

- **Statically typed languages:** all or almost all type checking occurs at compilation time.
 - C, Java, etc.
- **Dynamically typed languages:** almost all checking of types is done as part of program execution.
 - Scheme
- **Untyped languages:** no type checking
 - Assembly, Machine code

Static and Dynamic Types

- The **dynamic type** of an object is the class C that is used in the *new* $C()$ expression that construct the object.
 - Runtime notion
 - Even languages that are not statically typed have the notion of dynamic types
- The **static type** of an object is a notation that encapsulates all possible types the expression could take.
 - Compile-time notion

Type Inference (Crash Introduction to Natural Deduction)

- 1 Why using Types?
- 2 What is Type Checking?
- 3 Type Inference (Crash Introduction to Natural Deduction)
- 4 Type Checking in Practice
- 5 An Overview of Types

Type Checking and Type Inference

- Type Checking is the process of verifying fully typed programs
- Type Inference is the process of filling in missing type information
- The two are different, but are often used interchangeably!

Type Checking and Type Inference

- Type Checking is the process of verifying fully typed programs
- Type Inference is the process of filling in missing type information
- The two are different, but are often used interchangeably!

Type Checking and Type Inference

- Type Checking is the process of verifying fully typed programs
- Type Inference is the process of filling in missing type information
- The two are different, but are often used interchangeably!

- **Types not need to be explicit to have static typing.**
With the rule rules, one can infer types!
- We use an appropriate formalism to express inference rules!
 - Given a proper notation we can check the accuracy of the rules
 - Given a proper notation, we can easily translate it into programs.

From English to an Inference Rule

If e_1 has type Int and e_2 has type Int
then $e_1 + e_2$ has type Int .

$(e_1 \text{ has type } Int \wedge e_2 \text{ has type } Int)$
 $\implies e_1 + e_2 \text{ has type } Int.$

$(e_1: Int \wedge e_2: Int)$
 $\implies e_1 + e_2: Int.$

From English to an Inference Rule

If e_1 has type Int and e_2 has type Int
then $e_1 + e_2$ has type Int .

$(e_1 \text{ has type } Int \wedge e_2 \text{ has type } Int)$
 $\implies e_1 + e_2 \text{ has type } Int.$

$(e_1: Int \wedge e_2: Int)$
 $\implies e_1 + e_2: Int.$

From English to an Inference Rule

If e_1 has type Int and e_2 has type Int
then $e_1 + e_2$ has type Int .

$(e_1 \text{ has type } Int \wedge e_2 \text{ has type } Int)$
 $\implies e_1 + e_2 \text{ has type } Int.$

$(e_1: Int \wedge e_2: Int)$
 $\implies e_1 + e_2: Int.$

The statement:

- $(e_1: \text{Int} \wedge e_2: \text{Int}) \implies e_1 + e_2: \text{Int}.$

... is a special case of:

- $(\text{Hypothesis}_1: \text{Int} \wedge \dots \wedge \text{Hypothesis}_n: \text{Int}) \implies \text{Conclusion}$

This is an inference rule!

By tradition inferences rules are written:

$$\frac{\vdash \text{Hyp}_1 \dots \vdash \text{Hyp}_n}{\vdash \text{Conclusion}}$$

$\vdash$ means *is provable that ...*

Example

Detect the type of a variable:

$$\frac{\vdash i \text{ is an integer}}{\vdash i: \text{Int}}$$

Detect the type of an expression:

$$\frac{\vdash e_1: \text{Int} \wedge \vdash e_2: \text{Int}}{\vdash e_1 + e_2: \text{Int}}$$

Example

Detect the type of a variable:

$$\frac{\vdash i \text{ is an integer}}{\vdash i: \text{Int}}$$

Detect the type of an expression:

$$\frac{\vdash e_1: \text{Int} \wedge \vdash e_2: \text{Int}}{\vdash e_1 + e_2: \text{Int}}$$

Natural Deduction for Intuitionistic Logic

$$\overline{A_1, \dots, A_n \vdash A_k}$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \times B} \quad \frac{\Gamma \vdash A \times B}{\Gamma \vdash A} \quad \frac{\Gamma \vdash A \times B}{\Gamma \vdash B}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \quad \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A + B} \quad \frac{\Gamma \vdash B}{\Gamma \vdash A + B}$$

$$\frac{\Gamma \vdash A + B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}$$

Natural Deduction for Intuitionistic Logic

$$\frac{}{A_1, \dots, A_n \vdash A_k}$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \times B} \quad \frac{\Gamma \vdash A \times B}{\Gamma \vdash A} \quad \frac{\Gamma \vdash A \times B}{\Gamma \vdash B}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \quad \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A + B} \quad \frac{\Gamma \vdash B}{\Gamma \vdash A + B}$$

$$\frac{\Gamma \vdash A + B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}$$

Natural Deduction for Intuitionistic Logic

$$\frac{}{A_1, \dots, A_n \vdash A_k}$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \times B} \quad \frac{\Gamma \vdash A \times B}{\Gamma \vdash A} \quad \frac{\Gamma \vdash A \times B}{\Gamma \vdash B}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \quad \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A + B} \quad \frac{\Gamma \vdash B}{\Gamma \vdash A + B}$$

$$\frac{\Gamma \vdash A + B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}$$

Natural Deduction for Intuitionistic Logic

$$\frac{}{A_1, \dots, A_n \vdash A_k}$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \times B} \quad \frac{\Gamma \vdash A \times B}{\Gamma \vdash A} \quad \frac{\Gamma \vdash A \times B}{\Gamma \vdash B}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \quad \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A + B} \quad \frac{\Gamma \vdash B}{\Gamma \vdash A + B}$$

$$\frac{\Gamma \vdash A + B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}$$

Natural Deduction for Intuitionistic Logic

$$\begin{array}{c} \hline A_1, \dots, A_n \vdash A_k \\ \hline \end{array}$$
$$\begin{array}{c} \Gamma \vdash A \quad \Gamma \vdash B \quad \Gamma \vdash A \times B \quad \Gamma \vdash A \times B \\ \hline \Gamma \vdash A \times B \quad \Gamma \vdash A \quad \Gamma \vdash B \end{array}$$
$$\begin{array}{c} \Gamma, A \vdash B \quad \Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A \\ \hline \Gamma \vdash A \Rightarrow B \quad \Gamma \vdash B \end{array}$$
$$\begin{array}{c} \Gamma \vdash A \quad \Gamma \vdash B \\ \hline \Gamma \vdash A + B \quad \Gamma \vdash A + B \end{array}$$
$$\begin{array}{c} \Gamma \vdash A + B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C \\ \hline \Gamma \vdash C \end{array}$$

Natural Deduction for Intuitionistic Logic

$$\frac{}{x_1 : A_1, \dots, x_n : A_n \vdash x_k : A_k}$$

$$\frac{\Gamma \vdash u : A \quad \Gamma \vdash v : B}{\Gamma \vdash (u, v) : A \times B} \quad \frac{\Gamma \vdash u : A \times B}{\Gamma \vdash \text{fst}(u) : A} \quad \frac{\Gamma \vdash u : A \times B}{\Gamma \vdash \text{snd}(u) : B}$$

$$\frac{\Gamma, x : A \vdash u : B}{\Gamma \vdash \lambda x^A. u : A \Rightarrow B} \quad \frac{\Gamma \vdash f : A \Rightarrow B \quad \Gamma \vdash u : A}{\Gamma \vdash fu : B}$$

$$\frac{\Gamma \vdash u : A}{\Gamma \vdash \text{inl}^{A+B}(u) : A + B} \quad \frac{\Gamma \vdash u : B}{\Gamma \vdash \text{inr}^{A+B}(u) : A + B}$$

$$\frac{\Gamma \vdash w : A + B \quad \Gamma, x : A \vdash u : C \quad \Gamma, y : B \vdash v : C}{\Gamma \vdash \text{case } w \text{ of } \text{inl}(x).u \mid \text{inr}(y).v : C}$$

$$\frac{}{\Gamma \vdash n : \text{Int}} \quad \frac{}{\Gamma \vdash s : \text{String}}$$

$$\frac{}{x_1 : A_1, \dots, x_n : A_n \vdash x_k : A_k}$$

$$\frac{\Gamma \vdash a : \text{Int} \quad \Gamma \vdash b : \text{Int}}{\Gamma \vdash a + b : \text{Int}} + \quad \dots$$

$$\frac{\Gamma \vdash c : \text{Int} \quad \Gamma \vdash t : A \quad \Gamma \vdash f : A}{\Gamma \vdash \text{if } c \text{ then } t \text{ else } f : A} \text{ if then else}$$
$$\frac{\Gamma \vdash c : \text{Int} \quad \Gamma \vdash t : \text{Void}}{\Gamma \vdash \text{if } c \text{ then } t : \text{Void}} \text{ if then}$$

- It is a property of the type system
- Intuitively, a sound type system can correctly predict the type of a variable at runtime
- There can be many sound type rules, we need to use the most precise ones so it can be useful

Type Checking in Praticce

- 1 Why using Types?
- 2 What is Type Checking?
- 3 Type Inference (Crash Introduction to Natural Deduction)
- 4 Type Checking in Praticce
- 5 An Overview of Types

Type checking is done compositionally:

- 1 break down expressions into their subexpressions
- 2 type-check the subexpressions
- 3 ensure that the top-level compound expression can then be given a type itself

Throughout the process, a type environment is maintained which records the types of all variables in the expression.

Type Checking and functions

Functions:

- have types and can be used in expressions
- so they must be type-checked!

Main idea:

- 1 Look to the type of the body
- 2 Look to the type of the parameters
- 3 Deduce type for the function

For recursive functions (or types) the solution is to put all the headers in the environnement then put the bodies.

An Overview of Types

- 1 Why using Types?
- 2 What is Type Checking?
- 3 Type Inference (Crash Introduction to Natural Deduction)
- 4 Type Checking in Practice
- 5 An Overview of Types
 - Numeric Types
 - Other Primitive Types
 - Complex Types

Operation According to Types

Valid or invalid?

```
String s = "foobar"  
s = s + 12;
```

- In Java:
 - valid
 - Equivalent to: `s += String(12);`
- In C++:
 - a String is a `std::string`
 - invalid (we have to use `std::to_string`)
- In C
 - a String is a `char*`
 - Not really what we expect!

Operation According to Types

Valid or invalid?

```
String s = "foobar"  
s = s + 12;
```

- In **Java**:

- valid
- Equivalent to: `s += String(12);`

- In **C++**:

- a `String` is a `std::string`
- invalid (we have to use `std::to_string`)

- In **C**

- a `String` is a `char*`
- Not really what we expect!

Operation According to Types

Valid or invalid?

```
String s = "foobar"  
s = s + 12;
```

- In **Java**:
 - valid
 - Equivalent to: `s += String(12);`
- In **C++**:
 - a `String` is a `std::string`
 - invalid (we have to use `std::to_string`)
- In **C**
 - a `String` is a `char*`
 - Not really what we expect!

Operation According to Types

Valid or invalid?

```
String s = "foobar"  
s = s + 12;
```

- In **Java**:
 - valid
 - Equivalent to: `s += String(12);`
- In **C++**:
 - a `String` is a `std::string`
 - invalid (we have to use `std::to_string`)
- In **C**
 - a `String` is a `char*`
 - **Not really what we expect!**

Types and Binding

Checking types also requires to check iff associate operations are valid!

- One must define what are valid operations
- One must define what structure to represent a given type

Type checking and Binding are related!

Decreasing safety, increasing flexibility:

- during compilation (*static binding*)
- during loading
- when entering a subprogram
- when executing an instruction

Atomic, builtin

- Logical (Boolean)
- Numerical (integer, float, fixed, complex etc.)
- Character

User defined

- intervals
- enumerations
- arrays
- structures, records
- unions, variants

Atomic, builtin

- Logical (Boolean)
- Numerical (integer, float, fixed, complex etc.)
- Character

User defined

- intervals
- enumerations
- arrays
- structures, records
- unions, variants

Numeric Types

- 1 Why using Types?
- 2 What is Type Checking?
- 3 Type Inference (Crash Introduction to Natural Deduction)
- 4 Type Checking in Practice
- 5 An Overview of Types
 - Numeric Types
 - Other Primitive Types
 - Complex Types

Floats

- 1, sign
- 7, exponent
- 24, mantissa

Floats ANSI/IEEE Std 754-1985

[ANSI/IEEE, 1987, Goldberg, 1991]

IEEE Standard for Binary Floating Point Arithmetic

	FLT_	DBL_
RADIX	2	
MANT_DIG	24	53
DIG	6	15
MIN_EXP	-125	-1021
MIN_10_EXP	-37	-307
MAX_EXP	128	1024
MAX_10_EXP	+38	308
MIN	1.17549435E-38F	2.2250738585072014E-308
MAX	3.40282347E+38F	1.7976931348623157E+308
EPSILON	1.19209290E-07F	2.2204460492503131E-016

Floats are surprising (dangerous?)

foo.c

```
#include <stdio.h>

#define TEST(Op) \
    if ((num / den) Op quot) \
        puts("1 / 3 " #Op " 1 / 3");

int main()
{
    float quot = 1.0 / 3.0;
    volatile float num = 1, den = 3;

    TEST(<); TEST(<=);
    TEST(>); TEST(>=);
    TEST(==); TEST(!=);
}
```

Floats are surprising (dangerous?)

foo.c

```
#include <stdio.h>

#define TEST(Op) \
    if ((num / den) Op quot) \
        puts("1 / 3 " #Op " 1 / 3");

int main()
{
    float quot = 1.0 / 3.0;
    volatile float num = 1, den = 3;

    TEST(<); TEST(<=);
    TEST(>); TEST(>=);
    TEST(==); TEST(!=);
}
```

Optimized

```
% gcc-3.4 foo.c \
    -O1 -o foo-c
% ./foo-c
1 / 3 < 1 / 3
1 / 3 <= 1 / 3
1 / 3 != 1 / 3
```

Floats are surprising (dangerous?)

foo.c

```
#include <stdio.h>

#define TEST(Op) \
    if ((num / den) Op quot) \
        puts("1 / 3 " #Op " 1 / 3");

int main()
{
    float quot = 1.0 / 3.0;
    volatile float num = 1, den = 3;

    TEST(<); TEST(<=);
    TEST(>); TEST(>=);
    TEST(==); TEST(!=);
}
```

Optimized

```
% gcc-3.4 foo.c \
    -O1 -o foo-c
% ./foo-c
1 / 3 < 1 / 3
1 / 3 <= 1 / 3
1 / 3 != 1 / 3
```

Not Optimized

```
% gcc-3.4 foo.c \
    -o foo-c
% ./foo-c
1 / 3 > 1 / 3
1 / 3 >= 1 / 3
1 / 3 != 1 / 3
```

Floats are surprising (dangerous?)

From GCC's documentation

On 68000 and x86 systems, for instance, you can get paradoxical results if you test the precise values of floating point numbers. **For example, you can find that a floating point value which is not a NaN is not equal to itself.**

This results from the fact that the floating point registers hold a few more bits of precision than fit in a 'double' in memory. Compiled code moves values between memory and floating point registers at its convenience, and moving them into memory truncates them.

You can partially avoid this problem by using the `-ffloat-store` option.

Builtin Examples

- Looking at:
$$\frac{\vdash e_1 : \text{Int} \quad \vdash e_2 : \text{Float}}{\frac{e_1}{e_2} : \text{Float}}$$

- it seems that “Float” always wins ...

- but ...
$$\frac{\vdash e_1 : \text{Int}}{\text{int } k = e_1 : \text{Float}}$$

Builtin Examples

- Looking at:
$$\frac{\vdash e_1 : \text{Int} \quad \vdash e_2 : \text{Float}}{\frac{e_1}{e_2} : \text{Float}}$$

- it seems that “Float” always wins ...

- but ...
$$\frac{\vdash e_1 : \text{Int}}{\text{int } k = e_1 : \text{Float}}$$

Builtin Examples

- Looking at:
$$\frac{\vdash e_1 : \text{Int} \quad \vdash e_2 : \text{Float}}{\frac{e_1}{e_2} : \text{Float}}$$

- it seems that “Float” always wins ...

- but ...
$$\frac{\vdash e_1 : \text{Int}}{\text{int } k = e_1 : \text{Float}}$$

Fixed Point

- PL/I.

```
DECLARE X FIXED DECIMAL (6, 2);  
ON_CHECK(X): Y = X + 12;
```

- Essential in COBOL: “decimals are money”.

- Ada user defined precision.

```
type money is delta 0.01 range 0.0 .. 9999.99;
```

Were builtin at the beginning (FORTRAN).

Implies support for $+$, $-$, $*$, $/$

When designing a langage keep the builtin as small as possible!.

Other Primitive Types

- 1 Why using Types?
- 2 What is Type Checking?
- 3 Type Inference (Crash Introduction to Natural Deduction)
- 4 Type Checking in Practice
- 5 **An Overview of Types**
 - Numeric Types
 - **Other Primitive Types**
 - Complex Types

Primitive types

- Numeric
- Boolean (since ALGOL 60)
- Characters
- String (SNOBOL4 with Pattern-Matching)
 - Should strings be simply a special kind of character array or a primitive type?
 - Should strings have static or dynamic length?
 - What operations? copy? slices? affect? length?
 - Notion of descriptor

Primitive types

- Numeric
- Boolean (since ALGOL 60)
- Characters
- String (SNOBOL4 with Pattern-Matching)
 - Should strings be simply a special kind of character array or a primitive type?
 - Should strings have static or dynamic length?
 - What operations? copy? slices? affect? length?
 - Notion of descriptor

Primitive types

- Numeric
- Boolean (since ALGOL 60)
- Characters
- String (SNOBOL4 with Pattern-Matching)
 - Should strings be simply a special kind of character array or a primitive type?
 - Should strings have static or dynamic length?
 - What operations? copy? slices? affect? length?
 - Notion of descriptor

Primitive types

- Numeric
- Boolean (since ALGOL 60)
- Characters
- String (SNOBOL4 with Pattern-Matching)
 - Should strings be simply a special kind of character array or a primitive type?
 - Should strings have static or dynamic length?
 - What operations? copy? slices? affect? length?
 - Notion of descriptor

Complex Types

- 1 Why using Types?
- 2 What is Type Checking?
- 3 Type Inference (Crash Introduction to Natural Deduction)
- 4 Type Checking in Practice
- 5 An Overview of Types
 - Numeric Types
 - Other Primitive Types
 - Complex Types

Enumerations

- Integer ranges, or ordered labels

```
type days_num = 1 .. 7;
```

```
type days = (Monday, Tuesday, Wednesday, Thursday,  
             Friday, Saturday, Sunday);
```

- Are enumeration values coerced to integer?

- Conflicts are solved in Ada

```
type light is (red, orange, green);
```

```
type flag is (red, orange, green);
```

- Common operations

Pascal

```
pred (orange)
```

Ada

```
flag'pred(orange)
```

Enumerations

- Integer ranges, or ordered labels

```
type days_num = 1 .. 7;
```

```
type days = (Monday, Tuesday, Wednesday, Thursday,  
             Friday, Saturday, Sunday);
```

- Are enumeration values coerced to integer?

- Conflicts are solved in Ada

```
type light is (red, orange, green);
```

```
type flag is (red, orange, green);
```

- Common operations

Pascal

```
pred (orange)
```

Ada

```
flag'pred(orange)
```

Enumerations

- Integer ranges, or ordered labels

```
type days_num = 1 .. 7;
```

```
type days = (Monday, Tuesday, Wednesday, Thursday,  
             Friday, Saturday, Sunday);
```

- Are enumeration values coerced to integer?

- Conflicts are solved in Ada

```
type light is (red, orange, green);
```

```
type flag is (red, orange, green);
```

- Common operations

Pascal

```
pred (orange)
```

Ada

```
flag'pred(orange)
```

Enumerations

- Integer ranges, or ordered labels

```
type days_num = 1 .. 7;
```

```
type days = (Monday, Tuesday, Wednesday, Thursday,  
             Friday, Saturday, Sunday);
```

- Are enumeration values coerced to integer?

- Conflicts are solved in Ada

```
type light is (red, orange, green);
```

```
type flag is (red, orange, green);
```

- Common operations

Pascal

```
pred (orange)
```

Ada

```
flag'pred(orange)
```

Enumerations

- Characters are enumerations in Ada, contrary to Pascal, C, etc.

```
type digits is ('0', '1', '2', '3', '4',  
               '5', '6', '7', '8', '9');  
type odigits is ('0', '2', '4', '6', '8');
```

- Iteration over enumerations

```
for l in '0' .. '8' loop
```

- Disambiguation

```
for l in odigits'('0')..odigits'('8') loop
```

Enumerations

- Characters are enumerations in Ada, contrary to Pascal, C, etc.

```
type digits is ('0', '1', '2', '3', '4',  
               '5', '6', '7', '8', '9');  
type odigits is ('0', '2', '4', '6', '8');
```

- Iteration over enumerations

```
for l in '0' .. '8' loop
```

- Disambiguation

```
for l in odigits('0')..odigits('8') loop
```

Enumerations

- Characters are enumerations in Ada, contrary to Pascal, C, etc.

```
type digits is ('0', '1', '2', '3', '4',  
               '5', '6', '7', '8', '9');  
type odigits is ('0', '2', '4', '6', '8');
```

- Iteration over enumerations

```
for l in '0' .. '8' loop
```

- Disambiguation

```
for l in odigits'('0')..odigits'('8') loop
```


- Available since Autocodes.

- In Fortran [Sun Microsystems, 1996]:

- up to 7 dimensions Since FORTRAN IV, originally 3

- ```
REAL TA0(2,2,3,4,5,6,10)
```

- Fortran 2008 requires supports up to 15.

- free lower bounds

- ```
REAL A(3:5, 7, 3:5), B(0:2)
```

- character arrays

- ```
CHARACTER M(3,4)*7, V(9)*4
```

- Available since Autocodes.
- In Fortran [Sun Microsystems, 1996]:

up to 7 dimensions Since FORTRAN IV, originally 3

```
REAL TAO(2,2,3,4,5,6,10)
```

Fortran 2008 requires supports up to 15.

free lower bounds

```
REAL A(3:5, 7, 3:5), B(0:2)
```

character arrays

```
CHARACTER M(3,4)*7, V(9)*4
```

- Available since Autocodes.
  - In Fortran [Sun Microsystems, 1996]:
    - up to 7 dimensions Since FORTRAN IV, originally 3
- `REAL TA0(2,2,3,4,5,6,10)`
- Fortran 2008 requires supports up to 15.

free lower bounds

`REAL A(3:5, 7, 3:5), B(0:2)`

character arrays

`CHARACTER M(3,4)*7, V(9)*4`

- Available since Autocodes.
- In Fortran [Sun Microsystems, 1996]:
  - up to 7 dimensions Since FORTRAN IV, originally 3  
`REAL TA0(2,2,3,4,5,6,10)`  
Fortran 2008 requires supports up to 15.
  - free lower bounds  
`REAL A(3:5, 7, 3:5), B(0:2)`
  - character arrays  
`CHARACTER M(3,4)*7, V(9)*4`

- Available since Autocodes.
- In Fortran [Sun Microsystems, 1996]:
  - up to 7 dimensions Since FORTRAN IV, originally 3  
`REAL TA0(2,2,3,4,5,6,10)`  
Fortran 2008 requires supports up to 15.
  - free lower bounds  
`REAL A(3:5, 7, 3:5), B(0:2)`
  - character arrays  
`CHARACTER M(3,4)*7, V(9)*4`

- In Fortran [Sun Microsystems, 1996]:

```
CHARACTER BUF(10)
```

- C

```
char buf[10];
```

- Pascal: size is part of the type!

```
var buf : array [0 .. 9] of char;
```

- Ada

```
buf : array (0..9) of characters;
```

- C++ (2011)

```
std::array<char, 10> buf;
```

# Arrays [Sebesta, 2002]

- In Fortran [Sun Microsystems, 1996]:  
`CHARACTER BUF(10)`
- C  
`char buf[10];`
- Pascal: size is part of the type!  
`var buf : array [0 .. 9] of char;`
- Ada  
`buf : array (0..9) of characters;`
- C++ (2011)  
`std::array<char, 10> buf;`

- In Fortran [Sun Microsystems, 1996]:  
`CHARACTER BUF(10)`
- C  
`char buf[10];`
- Pascal: size is part of the type!  
`var buf : array [0 .. 9] of char;`
- Ada  
`buf : array (0..9) of characters;`
- C++ (2011)  
`std::array<char, 10> buf;`



- In Fortran [Sun Microsystems, 1996]:  
`CHARACTER BUF(10)`
- C  
`char buf[10];`
- Pascal: size is part of the type!  
`var buf : array [0 .. 9] of char;`
- Ada  
`buf : array (0..9) of characters;`
- C++ (2011)  
`std::array<char, 10> buf;`

# Arrays [Sebesta, 2002]

- In Fortran [Sun Microsystems, 1996]:  
`CHARACTER BUF(10)`
- C  
`char buf[10];`
- Pascal: size is part of the type!  
`var buf : array [0 .. 9] of char;`
- Ada  
`buf : array (0..9) of characters;`
- C++ (2011)  
`std::array<char, 10> buf;`

# Initializing Arrays [Sebesta, 2002]

- In Fortran:

```
INTEGER LIST(3)
DATA LIST /0, 3, 5/
```

- Ada

```
list : array (0..2) of INTEGER := (1, 3, 5);
conv : array (0..9) of INTEGER := (4 => 2,
 5 => 1,
 others => 0);
```

# Initializing Arrays

- C

```
int list[] = {42, 51, 96};
const char *const names[] = {"Foo", "Bar", "Baz", "Qux"};
```

- C99

```
int a[5] = { [3] = 29, 30, [1] = 15 }; // { 0, 15, 0, 29, 30 }
int whitespace[256]
 = { [' '] = 1, ['\t'] = 1, ['\h'] = 1,
 ['\f'] = 1, ['\n'] = 1, ['\r'] = 1 };
```

- GNU C

```
int widths[] = { [0 ... 9] = 1, [10 ... 99] = 2, [100] = 3 };
```

- C++ (2011)

```
std::array<int, 3> a1{ {1,2,3} }; // double-braces required
std::array<int, 3> a2 = {1, 2, 3}; // except after =
auto a3 = std::array<int, 3>{1, 2, 3};
```

# Operation on Arrays: APL [Sebesta, 2002]

$\phi V$  Reverse  $V$

$\phi M$  Reverse the columns of  $M$

$\theta M$  Reverse the rows of  $M$

$\emptyset M$  Transpose  $M$   
(symbol should be flipped vertically).

$\boxed{\div} M$  Invert  $M$

# Operation on Arrays: APL [Sebesta, 2002]

$\phi V$  Reverse  $V$

$\phi M$  Reverse the columns of  $M$

$\theta M$  Reverse the rows of  $M$

$\emptyset M$  Transpose  $M$   
(symbol should be flipped vertically).

$\boxed{\div} M$  Invert  $M$

# Operation on Arrays: APL [Sebesta, 2002]

$\phi V$  Reverse  $V$

$\phi M$  Reverse the columns of  $M$

$\theta M$  Reverse the rows of  $M$

$\emptyset M$  Transpose  $M$   
(symbol should be flipped vertically).

$\boxed{\div} M$  Invert  $M$

# Operation on Arrays: APL [Sebesta, 2002]

$\phi V$  Reverse  $V$

$\phi M$  Reverse the columns of  $M$

$\theta M$  Reverse the rows of  $M$

$\emptyset M$  Transpose  $M$   
(symbol should be flipped vertically).

$\boxed{\div} M$  Invert  $M$



# Operation on Arrays: APL [Sebesta, 2002]

$\phi V$  Reverse  $V$

$\phi M$  Reverse the columns of  $M$

$\theta M$  Reverse the rows of  $M$

$\emptyset M$  Transpose  $M$   
(symbol should be flipped vertically).

$\boxed{\div} M$  Invert  $M$

- Originally, references (no `&`, no arithmetics etc.).
- Dynamic memory allocation.

# Pointers Arithmetics

A C “feature”.

`strlen.c`

```
size_t strlen(const char *cp)
{
 size_t res = 0;
 for (; *cp; ++cp)
 ++res;
 return res;
}
```

# A Better strlen

## strlen.c

```
size_t strlen(const char *cp)
{
 const char *cp2 = cp;
 for (/* nothing. */; *cp2; ++cp2)
 continue;
 return cp2 - cp;
}
```

# An Even Better (?) strlen

```
size_t strlen(const char *str)
{
 size_t res = 0;
 for (uint32_t *i = (uint32_t *)str;; ++i)
 {
 if (!(*i & 0x000000ff)) return res;
 if (!(*i & 0x0000ff00)) return res + 1;
 if (!(*i & 0x00ff0000)) return res + 2;
 if (!(*i & 0xff000000)) return res + 3;
 res += 4;
 }
}
```

- Beware of endianness (here, little endian)
- Beware of alignment
- Beware of Valgrind
- Still four *if*, one can suffice

# An Even Better (?) strlen

```
size_t strlen(const char *str)
{
 size_t res = 0;
 for (uint32_t *i = (uint32_t *)str;; ++i)
 {
 if (!(*i & 0x000000ff)) return res;
 if (!(*i & 0x0000ff00)) return res + 1;
 if (!(*i & 0x00ff0000)) return res + 2;
 if (!(*i & 0xff000000)) return res + 3;
 res += 4;
 }
}
```

- Beware of endianness (here, little endian)
- Beware of alignment
- Beware of Valgrind
- Still four *if*, one can suffice

# An Even Better (?) strlen

```
size_t strlen(const char *str)
{
 size_t res = 0;
 for (uint32_t *i = (uint32_t *)str;; ++i)
 {
 if (!(*i & 0x000000ff)) return res;
 if (!(*i & 0x0000ff00)) return res + 1;
 if (!(*i & 0x00ff0000)) return res + 2;
 if (!(*i & 0xff000000)) return res + 3;
 res += 4;
 }
}
```

- Beware of endianness (here, little endian)
- Beware of alignment
- Beware of Valgrind
- Still four *if*, one can suffice

# An Even Better (?) strlen

```
size_t strlen(const char *str)
{
 size_t res = 0;
 for (uint32_t *i = (uint32_t *)str;; ++i)
 {
 if (!(*i & 0x000000ff)) return res;
 if (!(*i & 0x0000ff00)) return res + 1;
 if (!(*i & 0x00ff0000)) return res + 2;
 if (!(*i & 0xff000000)) return res + 3;
 res += 4;
 }
}
```

- Beware of endianness (here, little endian)
- Beware of alignment
- Beware of Valgrind
- Still four *if*, one can suffice



# An Even Better (?) strlen

```
size_t strlen(const char *str)
{
 size_t res = 0;
 for (uint32_t *i = (uint32_t *)str;; ++i)
 {
 if (!(*i & 0x000000ff)) return res;
 if (!(*i & 0x0000ff00)) return res + 1;
 if (!(*i & 0x00ff0000)) return res + 2;
 if (!(*i & 0xff000000)) return res + 3;
 res += 4;
 }
}
```

- Beware of endianness (here, little endian)
- Beware of alignment
- Beware of Valgrind
- Still four `if`, one can suffice

# GLibc's strlen

```
/* One character at a time, until aligned on longword. */
...
unsigned long int longword_ptr = (unsigned long int *) char_ptr;
unsigned long int himagic = 0x80808080L;
unsigned long int lomagic = 0x01010101L;
if (sizeof (unsigned long int) > 4) {
 himagic = ((himagic << 16) << 16) | himagic;
 lomagic = ((lomagic << 16) << 16) | lomagic;
}
for (;;) {
 unsigned long int longword = *longword_ptr++;
 if (((longword - lomagic) & ~longword & himagic) != 0) {
 /* Which of the bytes was the zero? If none of them were,
 it was a misfire; continue the search. */
 const char *cp = (const char *) (longword_ptr - 1);
 if (cp[0] == 0) return cp - str;
 ...
 }
}
```

# Pointers and Arrays in C

- Arrays are almost pointers in C
- $a[i] \rightsquigarrow *(a + i)$  (pointer arithmetic)
- $a + i = i + a$
- $a + i \rightsquigarrow a + i * \text{sizeof}(*a)$  (integer arithmetic)

array[index] vs. index[array]

```
#include <stdio.h>

int main()
{
 int foo[2] = { 51, 42 };
 int zero = 0;
 printf("%d, %d\n", zero[foo], 1[foo]);
 return 0;
}
```

# Pointers and Arrays in C

- Arrays are almost pointers in C
- $a[i] \rightsquigarrow *(a + i)$  (pointer arithmetic)
- $a + i = i + a$
- $a + i \rightsquigarrow a + i * \text{sizeof}(*a)$  (integer arithmetic)

array[index] vs. index[array]

```
#include <stdio.h>

int main()
{
 int foo[2] = { 51, 42 };
 int zero = 0;
 printf("%d, %d\n", zero[foo], 1[foo]);
 return 0;
}
```

# Pointers and Arrays in C

- Arrays are almost pointers in C
- $a[i] \rightsquigarrow *(a + i)$  (pointer arithmetic)
- $a + i = i + a$
- $a + i \rightsquigarrow a + i * \text{sizeof}(*a)$  (integer arithmetic)

array[index] vs. index[array]

```
#include <stdio.h>

int main()
{
 int foo[2] = { 51, 42 };
 int zero = 0;
 printf("%d, %d\n", zero[foo], 1[foo]);
 return 0;
}
```

# Pointers and Arrays in C

- Arrays are almost pointers in C
- $a[i] \rightsquigarrow *(a + i)$  (pointer arithmetic)
- $a + i = i + a$
- $a + i \rightsquigarrow a + i * \text{sizeof}(*a)$  (integer arithmetic)

array[index] vs. index[array]

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
 int foo[2] = { 51, 42 };
```

```
 int zero = 0;
```

```
 printf("%d, %d\n", zero[foo], 1[foo]);
```

```
 return 0;
```

```
}
```

# Pointers and Arrays in C

- Arrays are almost pointers in C
- $a[i] \rightsquigarrow *(a + i)$  (pointer arithmetic)
- $a + i = i + a$
- $a + i \rightsquigarrow a + i * \text{sizeof}(*a)$  (integer arithmetic)

## array[index] vs. index[array]

```
#include <stdio.h>

int main()
{
 int foo[2] = { 51, 42 };
 int zero = 0;
 printf("%d, %d\n", zero[foo], 1[foo]);
 return 0;
}
```

# Dynamic Arrays

- Difficult implementation: dynamic stack frame.
- Deferred to a later lecture dedicated to stack frames.



# Tuples

Typical of functional languages.

```
pair.hs
```

```
fst :: (a,b) -> a
```

```
fst (x,y) = x
```

```
snd :: (a,b) -> b
```

```
snd (x,y) = y
```

```
let type person =
 {
 first_name : string,
 last_name : string
 };
```

Inheritance/extension.

# Variants in Functional Language

Extremely useful for syntax trees [Anisko, 2003].

## ast.hs

```
data Exp = Const Int
 | Name String
 | Temp String
 | Binop { op :: Op, lhs :: Exp, rhs :: Exp }
 | Mem Exp
 | Call { fun :: Exp, arg :: [Exp] }
 | ESeq { stm :: Stm, exp :: Exp }
data Op = Add | Sub | Mul | Div | Mod | And | Or
```

# Variants in Imperative Language

Decent in Pascal.

variants.p [gpc, 2003]

type

EyeColorType = (Red, Green, Blue, Brown, Pink);

PersonRec = record

Age: Integer;

case EyeColor: EyeColorType of

Red, Green : (WearsGlasses: Boolean);

Blue, Brown: (LengthOfLashes: Integer);

end;

# Unions

Sort of raw variants.

shape.c

```
typedef enum shape_kind_e
{
 shape_square,
 shape_circle,
 ...
} shape_kind_t;

struct Circle;
struct Square;
...
```

```
typedef struct Shape
{
 shape_kind_t kind;
 union {
 Square s;
 Circle c;
 ...
 } u;
} Shape;
```

- Algol has function types.
- Additional meaning in high-order functional languages.

# Tuples and Functions

`curry` convert an uncurried function to a curried function

`uncurry` convert a curried function to a function on pairs

## curry.hs

```
curry :: ((a, b) -> c) -> a -> b -> c
curry f x y = f (x, y)
```

```
uncurry :: (a -> b -> c) -> ((a, b) -> c)
uncurry f p = f (fst p) (snd p)
```

- A means to define a *list of*  $X$ , where  $X$  is a type variable.



# Genericity in C++

shape.cc

```
template <typename C>
class Shape
{
public:
 Shape(C x, C y)
 : x_(x), y_(y)
 {}
 // ...
private:
 C x_, y_;
};

auto f = Shape<int>{1, 2};
```

# Genericity in Eiffel

shape.e

```
class SHAPE[COORDINATE]
```

```
feature
```

```
 xc, yc : COORDINATE ;
```

```
 set_x_y(x,y : COORDINATE) is
```

```
 do
```

```
 xc := x ;
```

```
 yc := y ;
```

```
 end ;
```

```
...
```

# Genericity in Eiffel

shape.e

```
class SHAPE[COORDINATE->NUMERIC]
```

```
feature
```

```
 xc, yc : COORDINATE ;
```

```
 set_x_y(x,y : COORDINATE) is
```

```
 do
```

```
 xc := x ;
```

```
 yc := y ;
```

```
 end ;
```

```
...
```

# Genericity in Haskell: Signature of Maybe

## Maybe.hs

```
module Maybe(
 isJust, isNothing,
 fromJust, fromMaybe, listToMaybe, maybeToList,
 catMaybes, mapMaybe,
 -- ...and what the Prelude exports
 Maybe(Nothing, Just),
 maybe) where

data Maybe a = Nothing | Just a

isJust, isNothing :: Maybe a -> Bool
fromJust :: Maybe a -> a
fromMaybe :: a -> Maybe a -> a
listToMaybe :: [a] -> Maybe a
maybeToList :: Maybe a -> [a]
catMaybes :: [Maybe a] -> [a]
mapMaybe :: (a -> Maybe b) -> [a] -> [b]
```

# Genericity in Haskell: Implementation of Maybe

## Maybe.hs

```
isJust :: Maybe a -> Bool
isJust (Just a) = True
isJust Nothing = False

isNothing :: Maybe a -> Bool
isNothing = not . isJust

fromJust :: Maybe a -> a
fromJust (Just a) = a
fromJust Nothing = error "Maybe.fromJust: Nothing"

fromMaybe :: a -> Maybe a -> a
fromMaybe d Nothing = d
fromMaybe d (Just a) = a
```

# Genericity in Haskell: Implementation of Maybe

## Maybe.hs

```
maybeToList :: Maybe a -> [a]
maybeToList Nothing = []
maybeToList (Just a) = [a]

listToMaybe :: [a] -> Maybe a
listToMaybe [] = Nothing
listToMaybe (a:_) = Just a

catMaybes :: [Maybe a] -> [a]
catMaybes ms = [m | Just m <- ms]

mapMaybe :: (a -> Maybe b) -> [a] -> [b]
mapMaybe f = catMaybes . map f
```

# A Grammar for Types

## Tiger Types [Appel, 1998]

```
⟨Type⟩ ::= "Int" | "String" | "Void" | "Nil"
 | "Array" ⟨Type⟩
 | "Record" ("Id" ⟨Type⟩)
 | "Class" ("Id" ⟨Type⟩)
 | "Function" ("Id" ⟨Type⟩)
 | "Method" ("Id" ⟨Type⟩)
 | "Name" ⟨String⟩
```

# Comparing two Types

## Equivalence

- by structure
- by name

Pascal report did not define “equivalent types”.

### equivalence.p

```
type link = ^cell;
var next : link;
 last : link;
 p : ^cell;
 q, r : ^cell;
```



# Comparing two Types for Identity

equivalence.tig

```
type original = { value : int }
type copy = { value : int }
type alias = original
```

```
compatibility.ada
```

```
subtype index1 is integer 1 .. 10;
type index2 is new integer 1 .. 10;
```

Subtypes are constrained types.



(2003).

*The GNU Pascal Compiler Manual.*

Free Software Foundation, 59 Temple Place – Suite 330, Boston, MA  
02111-1307 USA, 20030830 edition.



Anisko, R. (2003).

Havm.

<http://tiger.lrde.epita.fr/Havm>.



ANSI/IEEE (1987).

ANSI/IEEE std 854-1987 – IEEE standard for radix-independent  
floating-point arithmetic.



Appel, A. W. (1998).

*Modern Compiler Implementation in C, Java, ML.*

Cambridge University Press.



Goldberg, D. (1991).

What every computer scientist should know about floating-point arithmetic.

*ACM Computing Surveys (CSUR)*, 23(1):5–48.

[http://docs.sun.com/source/806-3568/ncg\\_goldberg.html](http://docs.sun.com/source/806-3568/ncg_goldberg.html).



Sebesta, R. W. (2002).

*Concepts of programming languages (5th ed.)*.

Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.



Sun Microsystems (1996).

Fortran 77 Language Reference (for f77 4.2).

<http://www.ictp.trieste.it/~manuals/programming/sun/fortran/f77rm/index.html>.