A Set of Tools to Teach Compiler Construction

Akim Demaille Roland Levillain Benoît Perrot

EPITA Research and Development Laboratory (LRDE)
Paris, France

13th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE)

Universidad Politécnica de Madrid, Spain – July 2, 2008

Context and Motivation

The School

- EPITA: a French engineering school teaching computer science to graduate students.
- Third year (among five) is dedicated to the core curriculum.
- Strong practical emphasis on projects.

The Needs

- Ten years ago, the school 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.

Context and Motivation

The School

- EPITA: a French engineering school teaching computer science to graduate students.
- Third year (among five) is dedicated to the core curriculum.
- Strong practical emphasis on projects.

The Needs

- Ten years ago, the school 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.

Context and Motivation

The School

- EPITA: a French engineering school teaching computer science to graduate students.
- Third year (among five) is dedicated to the core curriculum.
- Strong practical emphasis on projects.

The Needs

- Ten years ago, the school 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.

Goals

Aim

Compiler construction as a by-product

Complete Project: Specifications, implementation, documentation, testing, distribution.

Several iterations: 7 (optionally up to 11) steps, for 6 (resp. up to 9) 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
: 7 (optionally up to 11) steps, for 6 (resp. up to 9) 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
: 7 (optionally up to 11) steps, for 6 (resp. up to 9) 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: 7 (optionally up to 11) steps, for 6 (resp. up to 9) 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** 7 (optionally up to 11) steps, for 6 (resp. up to 9) 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, Subversion, etc.

Understanding Computers Compiler and languages are tightly related to computer architecture.

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

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, Subversion, etc.

Understanding Computers Compiler and languages are tightly related to computer architecture.

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

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, Subversion, etc.

Understanding Computers Compiler and languages are tightly related to computer architecture.

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

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, Subversion, etc.

Understanding Computers Compiler and languages are tightly related to computer architecture.

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

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, Subversion, etc.

Understanding Computers
: Compiler and languages are tightly related to computer architecture.

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

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. [Debray, 2002]

- But... Students interested in compiler construction should be given the opportunity to work on challenging, optional assignments.

Non Goal

Writing a Compiler Paradoxically!

Well, at least considered a secondary issue.

- Why?
- But...

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. [Debray, 2002]

- But... Students interested in compiler construction should be given the opportunity to work on challenging, optional assignments.

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. [Debray, 2002]

- But... Students interested in compiler construction should be given the opportunity to work on challenging, optional assignments.

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. [Debray, 2002]

- But... Students interested in compiler construction should be given the opportunity to work on challenging, optional assignments.

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. [Debray, 2002]

- But... Students interested in compiler construction should be given the opportunity to work on challenging, optional assignments.

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. [Debray, 2002]*

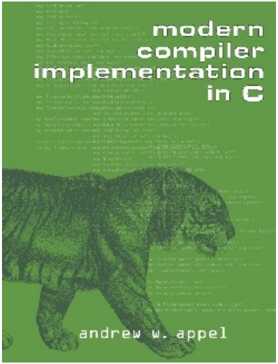
- But... Students interested in compiler construction should be given the opportunity to work on challenging, optional assignments.

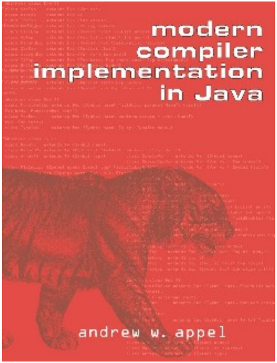
A Set of Tools to Teach Compiler Construction

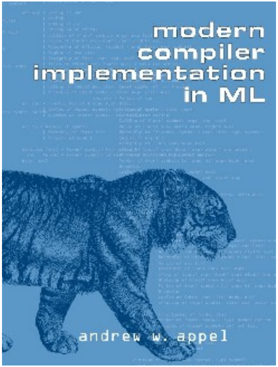
A Set of Tools to Teach Compiler Construction

A Compiler Construction Project Relevant to Core Curriculum

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







A Compiler Construction Project Relevant to Core Curriculum (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 Compiler Construction Project Relevant to Core Curriculum (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 Compiler Construction Project Relevant to Core Curriculum (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 Compiler Construction Project Relevant to Core Curriculum (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.

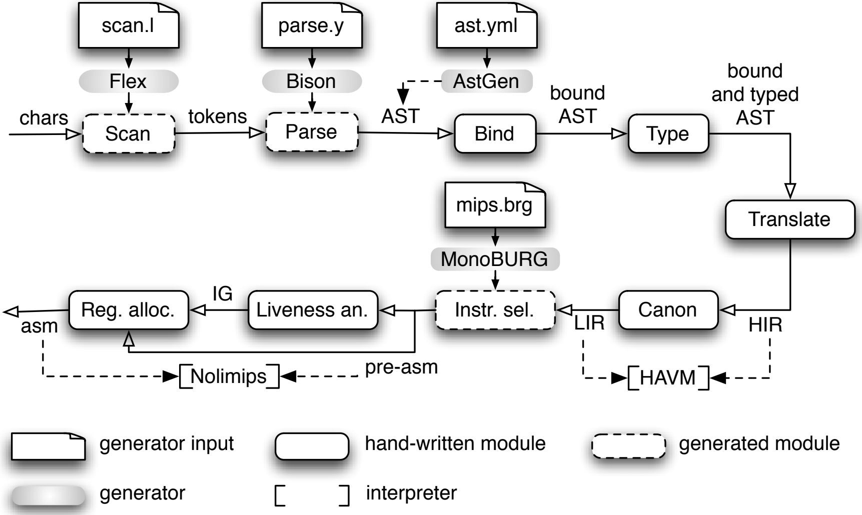
Project's *Modus Operandi*

- 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.

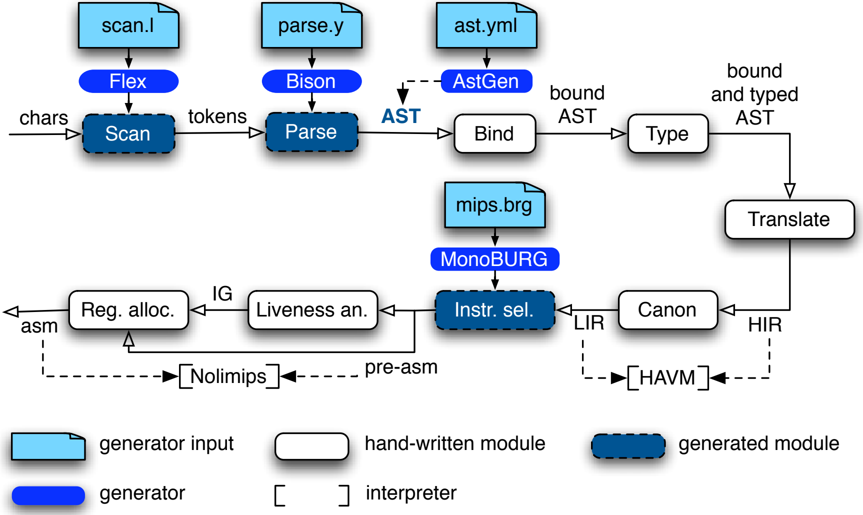
Project's *Modus Operandi*

- 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.

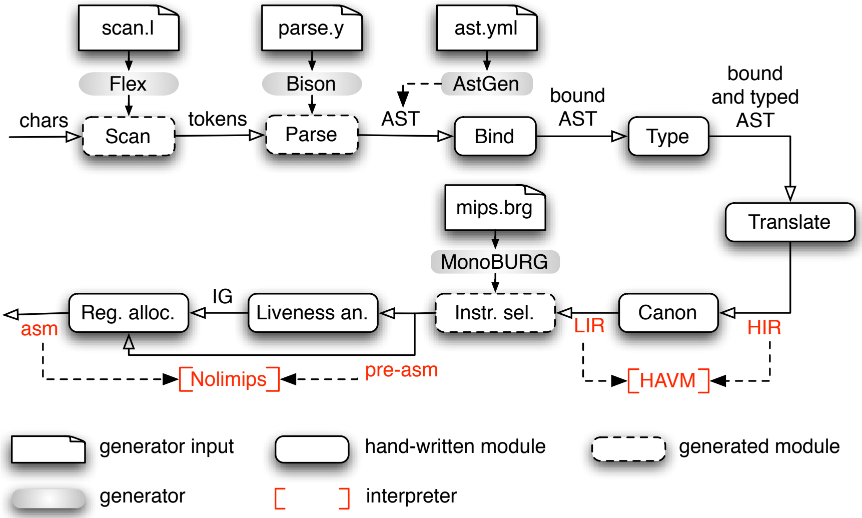
A Compiler as A Long Pipe



A Compiler as A Long Pipe



A Compiler as A Long Pipe



Figures

- 9 years of existence.
- 250 students per year (on average).
- Project done in groups of 4 (formerly 6) students.
- 7 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, 2007b] and project assignments [Demaille, 2007a]).

A Set of Tools to Teach Compiler Construction

Generated Components

- Many components of the Tiger compiler are generated.
- Some generators are provided to the students, others are for teachers' use only.

Generated components (for both students & teachers)

- The parser, generated by GNU Bison.
- The code generator, generated by MonoBURG.

Generated components (for teachers only)

- Data structures (classes) of the Abstract Syntax Tree (AST).
- Traversals (visitors) of the AST.

Generated Components

- Many components of the Tiger compiler are generated.
- Some generators are provided to the students, others are for teachers' use only.

Generated components (for both students & teachers)

- The parser, generated by GNU Bison.
- The code generator, generated by MonoBURG.

Generated components (for teachers only)

- Data structures (classes) of the AST.
- Traversals (visitors) of the AST.

Generated Components

- Many components of the Tiger compiler are generated.
- Some generators are provided to the students, others are for teachers' use only.

Generated components (for both students & teachers)

- The parser, generated by GNU Bison.
- The code generator, generated by MonoBURG.

Generated components (for teachers only)

- Data structures (classes) of the AST.
- Traversals (visitors) of the AST.

Rationale

Generated components are easier to

Understand They provide concise contents—for both students and teachers— with more semantics than bare source code.

Maintain The generation process tends to unify the output; moreover, it makes the selection of code hidden to students easier for the teachers.

Extend The conciseness of the input helps to focus on the added material.

Rationale

Generated components are easier to

Understand They provide concise contents—for both students and teachers— with more semantics than bare source code.

Maintain The generation process tends to unify the output; moreover, it makes the selection of code hidden to students easier for the teachers.

Extend The conciseness of the input helps to focus on the added material.

Rationale

Generated components are easier to

Understand They provide concise contents—for both students and teachers— with more semantics than bare source code.

Maintain The generation process tends to unify the output; moreover, it makes the selection of code hidden to students easier for the teachers.

Extend The conciseness of the input helps to focus on the added material.

Benefits for Students

- More time spent on the core of a topic for a given step,
  - e.g. writing and debugging a grammar (parser);
  - or understanding and applying pattern-based tree-rewriting principles (code generator).
- . . . Rather than on “side-effect issues”
  - memory allocation,
  - C++ idiosyncrasies,
  - lack of expressive power from the language in some areas,
  - etc.
- Those secondary-order problems are important though, and should be part of the assignments

→ Hence there are parts of the project that do not use generators.

Benefits for Students

- More time spent on the core of a topic for a given step,
  - e.g. writing and debugging a grammar (parser);
  - or understanding and applying pattern-based tree-rewriting principles (code generator).
- . . . Rather than on “side-effect issues”
  - memory allocation,
  - C++ idiosyncrasies,
  - lack of expressive power from the language in some areas,
  - etc.
- Those secondary-order problems are important though, and should be part of the assignments

→ Hence there are parts of the project that do not use generators.

Benefits for Students

- More time spent on the core of a topic for a given step,
  - e.g. writing and debugging a grammar (parser);
  - or understanding and applying pattern-based tree-rewriting principles (code generator).
- . . . Rather than on “side-effect issues”
  - memory allocation,
  - C++ idiosyncrasies,
  - lack of expressive power from the language in some areas,
  - etc.
- Those secondary-order problems are important though, and should be part of the assignments

→ Hence there are parts of the project that do not use generators.

Benefits for Teachers

- According to our experience the cost of using or even developing a generator is generally lower than writing everything by hand.
- For instance, the number of sources lines of codes of our AST description and its generators is less than the third of the lines of the generated C++ classes.
- Generators save us a huge amount of time when conducting experiments (renewing the project, adding optional assignments).

Benefits for Teachers

- According to our experience the cost of using or even developing a generator is generally lower than writing everything by hand.
- For instance, the number of sources lines of codes of our AST description and its generators is less than the third of the lines of the generated C++ classes.
- Generators save us a huge amount of time when conducting experiments (renewing the project, adding optional assignments).

Benefits for Teachers

- According to our experience the cost of using or even developing a generator is generally lower than writing everything by hand.
- For instance, the number of sources lines of codes of our AST description and its generators is less than the third of the lines of the generated C++ classes.
- Generators save us a huge amount of time when conducting experiments (renewing the project, adding optional assignments).

A Set of Tools to Teach Compiler Construction

Bison, the GNU Yacc implementation [Corbett et al., 2003]

- Free software (GNU GPL).
- Backward-compatible with Yacc.
- With many additions:
    - C++ and Java back-ends,
    - GLR algorithm in addition to LALR(1),
    - improved programming interface,
    - better debug tools.
- Used along with Flex (free software Lex implementation).

Our Contributions to Bison

- A LALR(1) C++ back-end.
- A GLR C++ back-end.
- Helpers to pretty-print symbols and manage memory during error handling.
- Improved debug information from Bison and the generated parser.
- Named symbols in productions.

Our Contributions to Bison

- A LALR(1) C++ back-end.
- A GLR C++ back-end.
- Helpers to pretty-print symbols and manage memory during error handling.
- Improved debug information from Bison and the generated parser.
- Named symbols in productions.

Our Contributions to Bison

- A LALR(1) C++ back-end.
- A GLR C++ back-end.
- Helpers to pretty-print symbols and manage memory during error handling.
- Improved debug information from Bison and the generated parser.
- Named symbols in productions.

Our Contributions to Bison

- A LALR(1) C++ back-end.
- A GLR C++ back-end.
- Helpers to pretty-print symbols and manage memory during error handling.
- Improved debug information from Bison and the generated parser.
- Named symbols in productions.

Our Contributions to Bison

- A LALR(1) C++ back-end.
- A GLR C++ back-end.
- Helpers to pretty-print symbols and manage memory during error handling.
- Improved debug information from Bison and the generated parser.
- Named symbols in productions.

Our Contributions to Bison

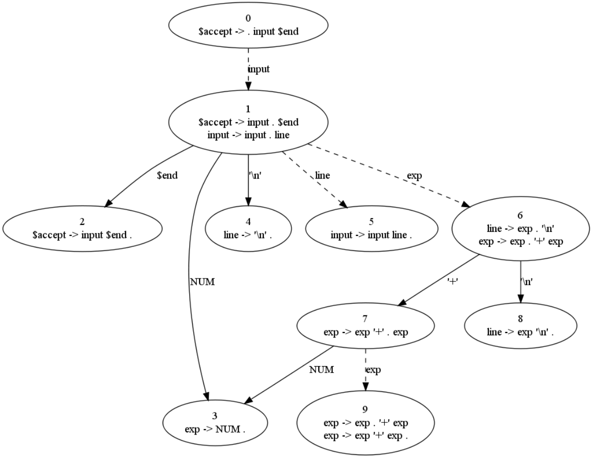
Debug Information from Bison.

- Textual (improved with initial automaton state, easier-to-read state labels, lookahead symbols).
- Graphical (Graphviz's `dot` format).

Our Contributions to Bison

Debug Information from Bison.

- Textual (improved with initial automaton state, easier-to-read state labels, lookahead symbols).
- Graphical (Graphviz's `dot` format).



Our Contributions to Bison

Named Symbols in Productions

- Before:

```
exp: "identifier" "[" exp "]" "of" exp
//$$      $1       $2  $3  $4   $5  $6
{ $$ = new Array (@$, new Type (@1, $1), $3, $6) }
```

- After:

```
exp: "identifier"$type "[" exp$size "]" "of" exp$init
{ $$ = new Array(@$, new Type(@type, $type), $size, $init) }
```

- Work in progress (should be integrated into the project next year).

Our Contributions to Bison

Named Symbols in Productions

- Before:

```
exp: "identifier" "[" exp "]" "of" exp
//$$      $1       $2  $3  $4   $5  $6
{ $$ = new Array (@$, new Type (@1, $1), $3, $6) }
```

- After:

```
exp: "identifier"$type "[" exp$size "]" "of" exp$init
{ $$ = new Array(@$, new Type(@type, $type), $size, $init) }
```

- Work in progress (should be integrated into the project next year).

Assignments and Results

First stage Students have to write an LALR(1) parser (along with a scanner generated by Flex) from scratch using Bison and its C++ back-end.

Second stage Students convert their LALR(1) parser to a GLR one (e.g., they can simplify the grammar to accept local ambiguities).

Results

- Students debug their parser easier and faster than before, thanks to Bison's helpful information.
- They learn more material from language theory (GLR, for instance) in the same amount of time.

Assignments and Results

First stage Students have to write an LALR(1) parser (along with a scanner generated by Flex) from scratch using Bison and its C++ back-end.

Second stage Students convert their LALR(1) parser to a GLR one (e.g., they can simplify the grammar to accept local ambiguities).

Results

- Students debug their parser easier and faster than before, thanks to Bison's helpful information.
- They learn more material from language theory (GLR, for instance) in the same amount of time.

Assignments and Results

First stage Students have to write an LALR(1) parser (along with a scanner generated by Flex) from scratch using Bison and its C++ back-end.

Second stage Students convert their LALR(1) parser to a GLR one (e.g., they can simplify the grammar to accept local ambiguities).

Results

- Students debug their parser easier and faster than before, thanks to Bison's helpful information.
- They learn more material from language theory (GLR, for instance) in the same amount of time.

A Set of Tools to Teach Compiler Construction

Why Generating the Abstract Syntax Tree and its Traversals?

- From the input (concrete syntax) the parser generates an AST.
- Usually, in Object-Oriented Language:
  - ASTs node types are implemented as “uniform” classes.
  - Tree traversals are based on the VISITOR design pattern.
- Hence both might be generated.

A Teacher-Only Generator

- Writing AST and traversal is part of the first C++ assignment of the project, that students are expected to write.

→ This part of the compiler is actually generated, but

  - students only see the output, with gaps;
  - the AST description (input) and the generators are kept private.

- Input is a simple YAML description, generators are written in Python.

A Teacher-Only Generator

- Writing AST and traversal is part of the first C++ assignment of the project, that students are expected to write.

→ This part of the compiler is actually generated, but
  - students only see the output, with gaps;
  - the AST description (input) and the generators are kept private.
- Input is a simple YAML description, generators are written in Python.

A Teacher-Only Generator

- Writing AST and traversal is part of the first C++ assignment of the project, that students are expected to write.

→ This part of the compiler is actually generated, but
  - students only see the output, with gaps;
  - the AST description (input) and the generators are kept private.
- Input is a simple YAML description, generators are written in Python.

Generating the AST

YAML description

```yaml
---
# Top-most class of the AST class hierarchy.
Ast:
  attributes:
    - location:
type: Location
desc: Scanner position information

Dec:
  super: Ast Typable
  attributes:
    - name:
type: Symbol
desc: Name of the defined entity
owned: False
```

C++ code generated

```cpp
class Dec: public Ast, public Typable {
 public:
  Dec (const Location& l, const Symbol& n);
  virtual ~Dec ();

  /** \name Visitors entry point.  \{ */
  /// Accept a const visitor \a v.
  virtual void accept (ConstVisitor& v) const = 0;
  /// Accept a non-const visitor \a v.
  virtual void accept (Visitor& v) = 0;
  /** \} */

  /** \name Accessors.  \{ */
  /// Return name of the defined entity.
  const Symbol& name_get () const;
  /// Set name of the defined entity.
  void name_set (const Symbol&);
  /** \} */

protected:
  /// Name of the defined entity.
  Symbol name_;
};
```

Benefits

- Architecture (as well as gaps in the code) are easy to change from year to year.

$\rightarrow$ Discourages teaching by stealing code from previous years.

- Language changes/extensions are made easier and faster.
- For instance, adding AST classes and basic visitors (abstract, identity, cloning and pretty-printing visitors) for OO constructs took only 50 lines of YAML description.

A Set of Tools to Teach Compiler Construction

Parsing vs Code Generation

- Code generation (aka instruction selection) is symmetric to parsing.

	Parsing	Code generation
Input	(Tiger) Code	IR trees
Tool	Parser	Code generator
Output	AST	Assembly language code

- Both task make use of a grammar to match their entries.
- Yet another part to be generated.
- Existing literature and tools (Twig [Appel, 1987], BURG [Fraser et al., 1991]).
- Again, existing tools did no suit our plans.

Parsing vs Code Generation

- Code generation (aka instruction selection) is symmetric to parsing.

	Parsing	Code generation
Input	(Tiger) Code	IR trees
Tool	Parser	Code generator
Output	AST	Assembly language code

- Both task make use of a grammar to match their entries.
- Yet another part to be generated.
- Existing literature and tools (Twig [Appel, 1987], BURG [Fraser et al., 1991]).
- Again, existing tools did no suit our plans.

MonoBURG

- We extended MonoBURG, from the Mono Project.
- C++ features (namespaces, references).
- Named arguments in rules.
- Modules (`%include` directive).
- Better debugging (`#line` statements in emitted code).

Benefits

- Helped students write their code generators.
- Allowed some of them to write a few optimizations.
- Allowed us to provide a second back-end (IA-32) in addition to the first one (MIPS).
- Writing a tree pattern matcher by hand in C++ is a tedious task.
- Less interesting than focusing on the tree rewriting task itself.

A Set of Tools to Teach Compiler Construction

Checking Outputs of the Compiler

- Many deliveries (7, up to 11) from students, all to be automatically checked.
- Compiler produces outputs at almost every stage: pretty-printed AST (possibly annotated), intermediate representation, code generation, etc.
- Interpreters are needed
  - for students, to check their compiler during development;
  - for teachers, to assess the work of students.
- Two interpreters have been developed

  Havm A virtual machine interpreting code from the intermediate representation (middle-end).

  Nolimips A simulator executing MIPS code (back-end).
- Both are free software and are available on the Internet.

Checking Outputs of the Compiler

- Many deliveries (7, up to 11) from students, all to be automatically checked.
- Compiler produces outputs at almost every stage: pretty-printed AST (possibly annotated), intermediate representation, code generation, etc.
- Interpreters are needed
  - for students, to check their compiler during development;
  - for teachers, to assess the work of students.
- Two interpreters have been developed

  Havm A virtual machine interpreting code from the intermediate representation (middle-end).

  Nolimips A simulator executing MIPS code (back-end).
- Both are free software and are available on the Internet.

Checking Outputs of the Compiler

- Many deliveries (7, up to 11) from students, all to be automatically checked.
- Compiler produces outputs at almost every stage: pretty-printed AST (possibly annotated), intermediate representation, code generation, etc.
- Interpreters are needed
  - for students, to check their compiler during development;
  - for teachers, to assess the work of students.
- Two interpreters have been developed
  - **Havm** A virtual machine interpreting code from the intermediate representation (middle-end).
  - **Nolimips** A simulator executing MIPS code (back-end).
- Both are free software and are available on the Internet.

A Set of Tools to Teach Compiler Construction

Assessment of the Middle-End

- Checking the Intermediate Representation (IR) is troublesome before completion of the compiler.
- Evaluation done by executing the IR.
- The IR from the Tiger compiler uses the TREE language, a high-level register-based Intermediate Language (IL).
- No tool available.

→ Havm, a TREE interpreter.

Assessment of the Middle-End

- Checking the Intermediate Representation (IR) is troublesome before completion of the compiler.
- Evaluation done by executing the IR.
- The IR from the Tiger compiler uses the TREE language, a high-level register-based IL.
- No tool available.

→ Havm, a TREE interpreter.

Assessment of the Middle-End

- Checking the Intermediate Representation (IR) is troublesome before completion of the compiler.
- Evaluation done by executing the IR.
- The IR from the Tiger compiler uses the TREE language, a high-level register-based IL.
- No tool available.

→ Havm, a TREE interpreter.

Havm

- Free students from handling the stack (implicit support for recursion).
- Support for an unlimited set of temporaries (pseudo-registers).
- Trace mode.
- Performance measurement (profiling).
- Havm proved to be a valuable tool to help student develop and understand the middle-end, and improve it.

A Set of Tools to Teach Compiler Construction

MIPS

- MIPS is simple and elegant, perfectly fitted for education.
- However, MIPS is not necessarily common hardware in a computer science school.
- Moreover, a MIPS machine would not be able to execute code without register allocation.

→ Use a simulator.

Nolimips

- Existing simulators (SPIM [Larus, 1990], MARS [Vollmar and Sanderson, 2006]) are good, but lack some features.
- Nolimips, a new MIPS simulator.
  - Can execute code using an arbitrary number of registers.
  - Can up- or downgrade the MIPS architecture by increasing/decreasing the number of registers.
  - Provides with a small set of system calls (I/O, memory management).
  - Trace mode.
  - Interactive shell.
- A useful tool to diagnose mistakes and debug the back-end of the compiler.

Nolimips

- Existing simulators (SPIM [Larus, 1990], MARS [Vollmar and Sanderson, 2006]) are good, but lack some features.
- Nolimips, a new MIPS simulator.
  - Can execute code using an arbitrary number of registers.
  - Can up- or downgrade the MIPS architecture by increasing/decreasing the number of registers.
  - Provides with a small set of system calls (I/O, memory management).
  - Trace mode.
  - Interactive shell.
- A useful tool to diagnose mistakes and debug the back-end of the compiler.

A Set of Tools to Teach Compiler Construction

Other Tools

The Tiger project also features (teacher) tools for:

Generating code with gaps Based on annotations in either code or description of components.

Automating deliveries Students just have to upload their work, which get a timestamp and is possibly immediately evaluated.

Automating evaluation Runs a big (private) test suite, and makes use of the reference compiler and interpreters (Havm & Nolimips).

Interactive compiler sessions The Tiger Compiler Shell: a Python- or Ruby-based shell giving access to the compiler's component.

Feedback From the Users

- According to teaching assistants, interaction with students is less demanding from year to year.
- Students are aware of the flaws of their work
  - either because they used the tools and discovered them themselves,
  - or because we (teachers/assistants) pointed them at errors (interpreters are used to grade and generate automated post-delivery reports to students).

Feedback from the Teachers

- Quality of the delivered compiler increased.
  From one year to the following,
  - Groups whose parser passes more than 97% of the tests $\times 2$
  - Number of correct ASTs $\times 3$
  - Average grade for the binding stage $\times 2$
- Automating is a gain of time, allowing teacher and assistants to work on other, more advanced issues.

Frequent Reports from Alumni

- "I understood those design patterns thanks to the Tiger project (and I'm using them right now)."
- "I'm usually the one being asked questions about C++/design/development/tools in my professional environment thanks to what I learned during this project."
- "I had a problem last week, and I remembered (and reused!) a solution from my own instance of the Tiger project."

Conclusion and Future Work

- Tools in a CS programming assignment are profitable.
- We highly encourage to either use, extend or even develop them!
- Some presented here are already available.
- Contact us about the others.
- We would like to share experience and tools on the Tiger Project.

`http://tiger.lrde.epita.fr`
tiger@lrde.epita.fr

A Set of Tools to Teach Compiler Construction

1. The Tiger Project
2. Compiler Components Generation
   - Parser Generation
   - Abstract Syntax Tree and Traversals Generation
   - Code Generator Generation
3. Pedagogical Interpreters
   - Register-based Intermediate Language
   - MIPS Assembly Language
4. Results and discussion

`http://tiger.lrde.epita.fr`

tiger@lrde.epita.fr

History

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 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 (cont.)

2005 A second teacher in the project maintenance and supervision.
First uses of some Boost library (Boost.Variant, Boost Graph Library (BGL), Smart Pointers).
First use of concrete-syntax program transformations (code generation)

2007 Tiger becomes an Object-Oriented Language (OOL).
Full concrete-syntax rewriting engine (code matching & generation).

2008 Extension of Bison's grammar to handle named parameters (pending).

Bibliography I

Appel, A. W. (1987).
Concise specifications of locally optimal code generators.
Technical Report CS-TR-080-87, Princeton University, Dept. of Computer Science, Princeton, New Jersey.

Appel, A. W. (1998).
Modern Compiler Implementation in C, Java, ML.
Cambridge University Press.

Corbett, R., Stallman, R., and Hilfinger, P. (2003).
Bison: GNU LALR(1) and GLR parser generator.
http://www.gnu.org/software/bison/bison.html.

Bibliography II

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.

Bibliography III

Demaille, A. (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. (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.

Fraser, C. W., Henry, R. R., and Proebsting, T. A. (1991).
BURG–fast optimal instruction selection and tree parsing.
Technical Report CS-TR-1991-1066.

Bibliography IV

Larus, J. R. (1990).
SPIM S20: A MIPS R2000 simulator.
Technical Report TR966, Computer Sciences Department, University of Wisconsin–Madison.

Vollmar, K. and Sanderson, P. (2006).
MARS: An education-oriented MIPS assembly language simulator.
In *Proceedings of the 37th SIGCSE technical symposium on Computer science education (SIGCSE'06)*, pages 239–243, Houston, Texas, USA. ACM Press.