$P4_{16}$ reference compiler implementation architecture

June 2021

Mihai Budiu (mbudiu-vmw)

mbudiu@vmware.com

What is this?

- A compiler for P4$_{16}$
- P4 = a language for programmable networks; see http://p4.org
- Compiles both P4$_{14}$ (i.e., P4 v1.0 and P4 v1.1) and P4$_{16}$ programs
- P4$_{16}$ specification: https://github.com/p4lang/p4-spec/tree/master/p4-16/spec
- Apache 2 license, open-source, reference implementation
- http//github.com/p4lang/p4c

Compiler goals

- Support current and future versions of P4
- Support multiple back-ends
  - Generate code for ASICs, NICs, FPGAs, software switches and other targets
- Provide support for other tools (debuggers, IDEs, control-plane, etc.)
- Open-source front-end
- Extensible architecture (easy to add new passes and optimizations)
- Use modern compiler techniques
  (immutable IR, visitor patterns, strong type checking, etc.)
- Comprehensive testing

What’s in the box

- Compiler source code (C++)
    - currently alpha quality release
- Two front-ends
    - $P4_{14}$ (v1.0, v1.1)
    - $P4_{16}$
- Converter $P4_{14}$ => $P4_{16}$
- Multiple back-ends:
    - eBPF => generates C code that can be compiled to extended Berkeley Packet Filters programs
    - uBPF => C code that can be compiled to user-space BPF
    - bmv2 => generates JSON files that can be used to drive the simple_switch network simulator built using BMv2 (behavioral model version 2)
    - p4test => fake test back-end
    - p4c-dpdk => generates DPDK assembly code to run in user-space
    - bmv2 psa => generates JSON for the PSA network simulator using BMv2

Example usage

- To pretty-print and validate a $P4_{16}$ file
  `p4test --pp out.p4 file.p4`
- To convert a $P4_{14}$ file to $P4_{16}$
  `p4test --pp out.p4 --std p4-14 file.p4`
- To compile a $P4_{14}$ file for the BMv2 simulator:
  `p4c-bm2-ss -o file.json --std p4-14 file.p4`
- To compile a P4 file for EBPF (via C):
  `p4c-ebpf –o file.c file.p4`

A fragment of the output

```
./p4c-bm2-ss: Compile a P4 program
--help                  Print this help message
--version               Print compiler version
-I path                 Specify include path (passed to preprocessor)
-D arg=value            Define macro (passed to preprocessor)
-U arg                  Undefine macro (passed to preprocessor)
-E                      Preprocess only, do not compile (prints program on stdout)
--nocpp                 Skip preprocess, assume input file is already preprocessed.
--std {14|16}           Specify language version to compile
--target target         Compile for the specified target
--arch arch             Compile for the specified architecture.
--pp file               Pretty-print the program in the specified file
--toJSON file           Dump IR to JSON in the specified file.
--p4runtime-file file   Write a control-plane API description to the specified file.
--p4runtime-entres-file file  Write static table entries as a P4Runtime WriteRequest message to the specified file.
--p4runtime-format f    Chose output format, one of {binary,json,text}.
-o outfile              Write output to outfile
--Wdisable[=diagnostic] Disable a compiler diagnostic, or disable all warnings
--Werror                Treat all warnings as errors.
-T loglevel             [Compiler debugging] Adjust logging level per file (see below)
-v                      [Compiler debugging] Increase verbosity level (can be repeated)
--top4 pass1[,pass2]    [Compiler debugging] Dump the P4 representation after
                        passes whose name contains one of `passX' substrings.
                        When '-v' is used this will include the compiler IR.
--dump folder           [Compiler debugging] Folder where P4 programs are dumped
--emit-externs          [BMv2 back-end] Force unknown externs to be emitted in the back-end.
```

How do I get started writing compiler code?

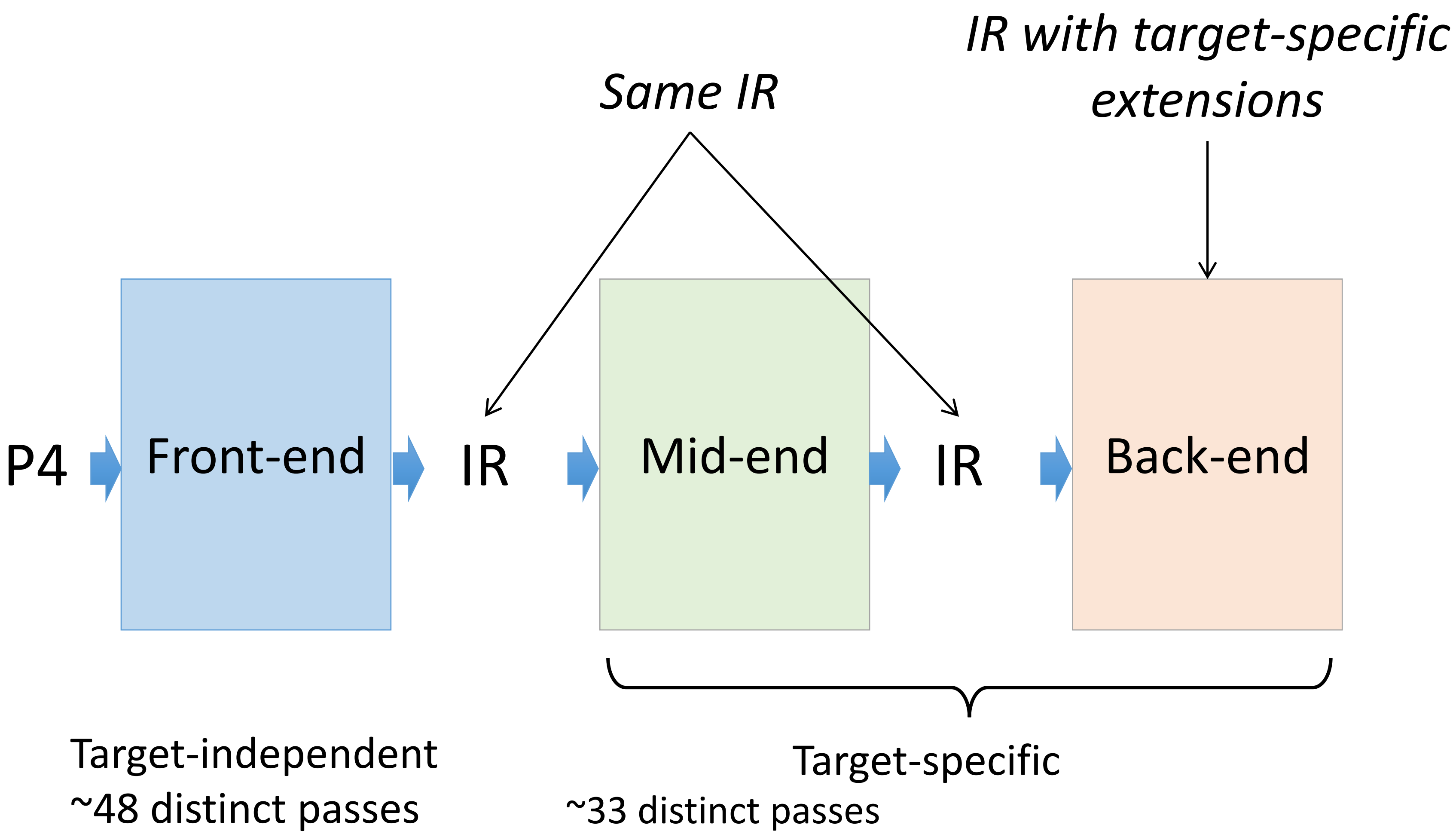
- Read the $P4_{16}$ spec
- Browse the *.def IR definition files and understand what they represent
- Understand the visitor interfaces (Inspector, Transform)
- Read the documentation to know what tools are available
  - The compiler doxygen documentation (still incomplete)
  - This document, especially the section “IR and Visitors”
- Browse the code top-down (starting from main)

Presentation Outline

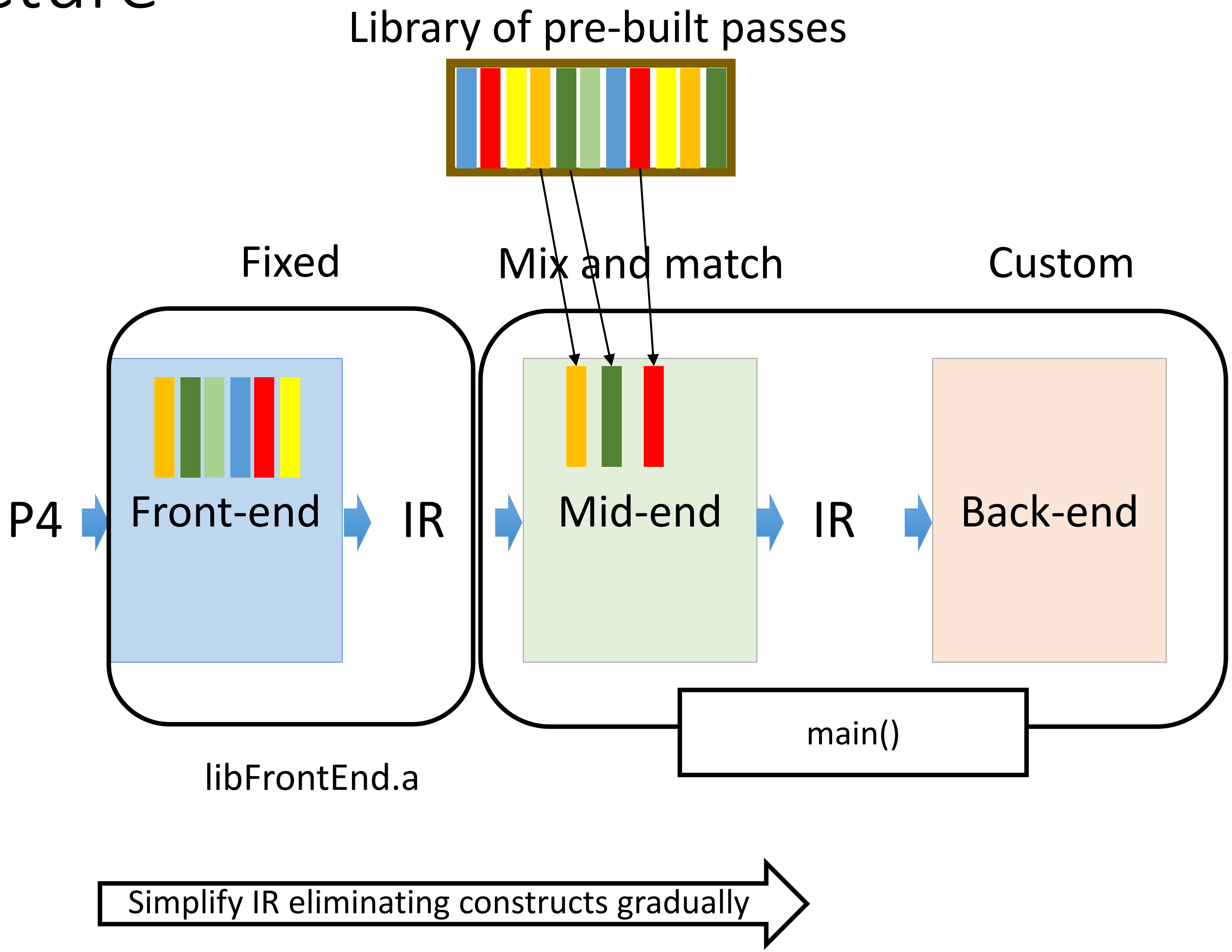
- Compiler architecture
- Compiler source code organization
- IR and visitors
- A guide to the provided passes
  - Front-end passes
  - Mid-end passes
- Sample back-ends

Compiler architecture

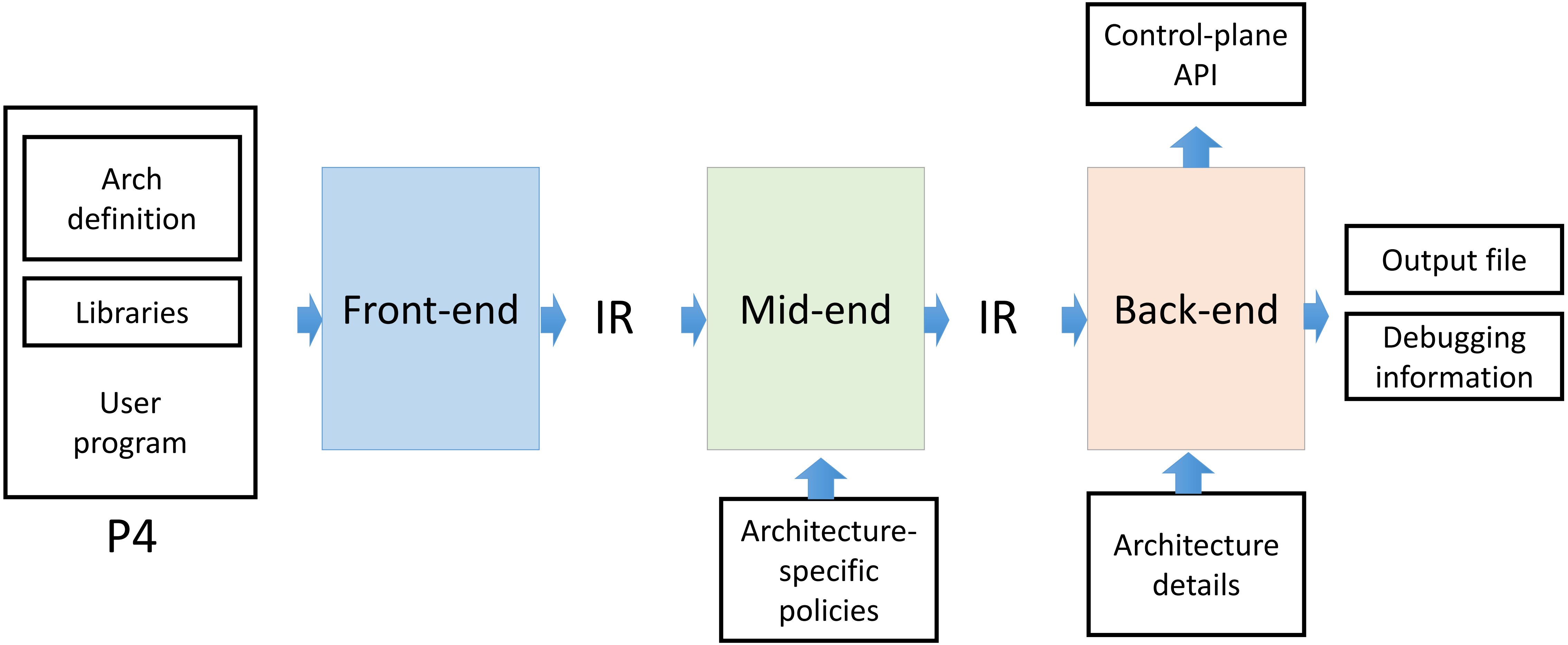
Compiler structure



Structure


Library of pre-built passes
Fixed
Mix and match
Custom
P4
Front-end
IR
Mid-end
IR
Back-end
main()
libFrontEnd.a
Simplify IR eliminating constructs gradually

Compiler flow



Compilation stages

- Front-end:
  - Completely architecture-independent
  - Program validation, type checking
  - Architecture-independent lowering and optimizations
- Mid-end:
  - architecture independent optimizations driven by architecture-dependent policies
  - Same base IR as front-end
- Back-end:
  - Completely target-dependent
  - Resource allocation, code generation
  - Can use a custom IR

Front-end passes

- Program parsing
- Validation
- Name resolution
- Type checking/type inference (Hindley-Milner)
- Make semantics explicit (e.g. order side-effects)
- Optimizations
- Inlining
- Compile-time evaluation & specialization
- Conversion to P4 source
- Deparser inference (for $P4_{14}$ programs)

After the front-end the control-plane API is generated

Mid-end passes

- Mid-end is different for each target
- Assembled from a library of existing passes
  - Optimizations
  - Create actions / tables from statements and actions
  - Eliminate tuple and enum types
  - Predicate code (convert ifs to ?:)
  - Etc.

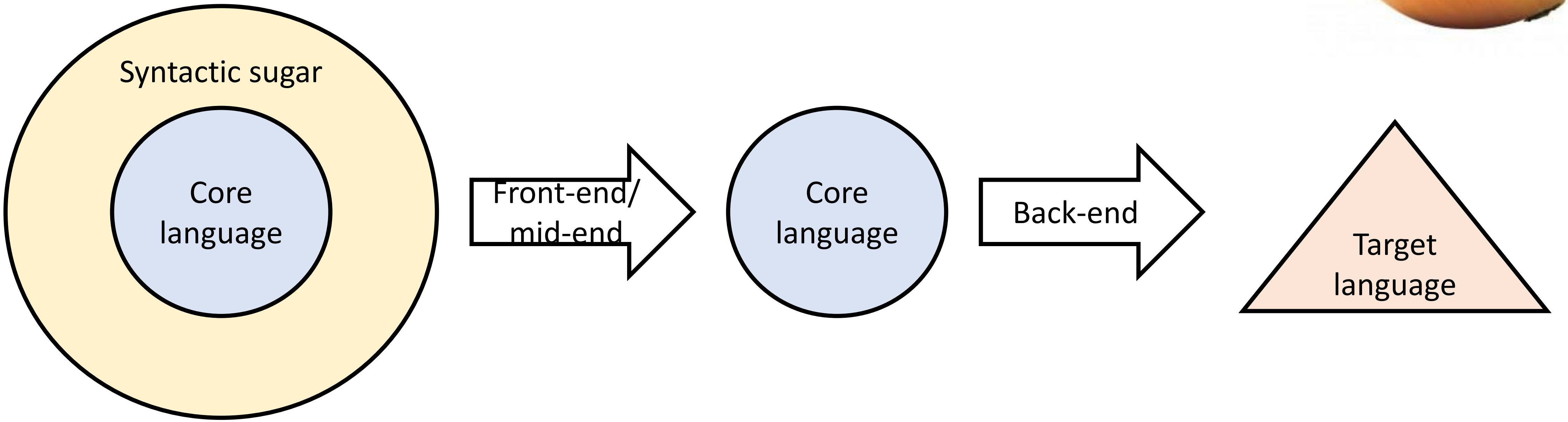
Back-end passes

- Target-specific
- Can backtrack, even back into mid-end
  (allows early passes to remake bad decisions)
- Lower code further to remove idioms not supported by target
- Resource allocation
  - Table allocation and placement
  - Register allocation
  - Parser timing and control
  - Allocate “extern” resources
- Target specific optimizations
- Code generation

Implementation details

- Common infrastructure for all compiler passes
  - Same IR and visitor base classes
  - Common utilities (error reporting, collections, strings, etc.)
- C++11, using garbage-collection (-lgc)
- Clean separation between front-end, mid-end and back-end
  - New mid+back-ends can be added easily
- IR can be extended (front-end and back-end may have different IRs)
- IR can be serialized to/from JSON
- Passes can be added easily

P4 Language Layered Design



- Many language constructs are eliminated entirely in the front-end/mid-end
- Syntactic sugar constructs are thus automatically supported by all back-ends

Additional documentation

- All documentation is in the source tree
- Source files are commented with doxygen
- Root of the documentation is in the docs/ folder of the source tree
- Provides links to other documentation files
- Each back-end can have additional documentation

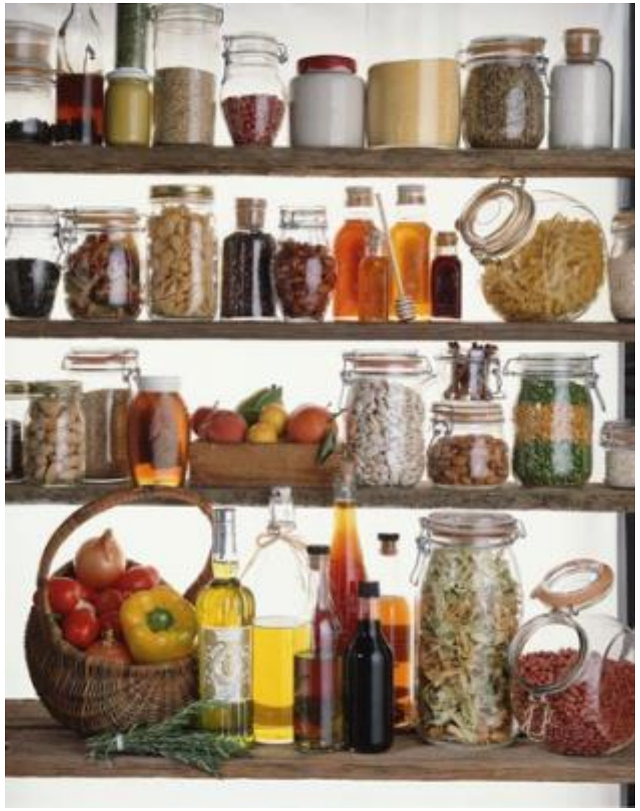
Source Code Organization

Repository

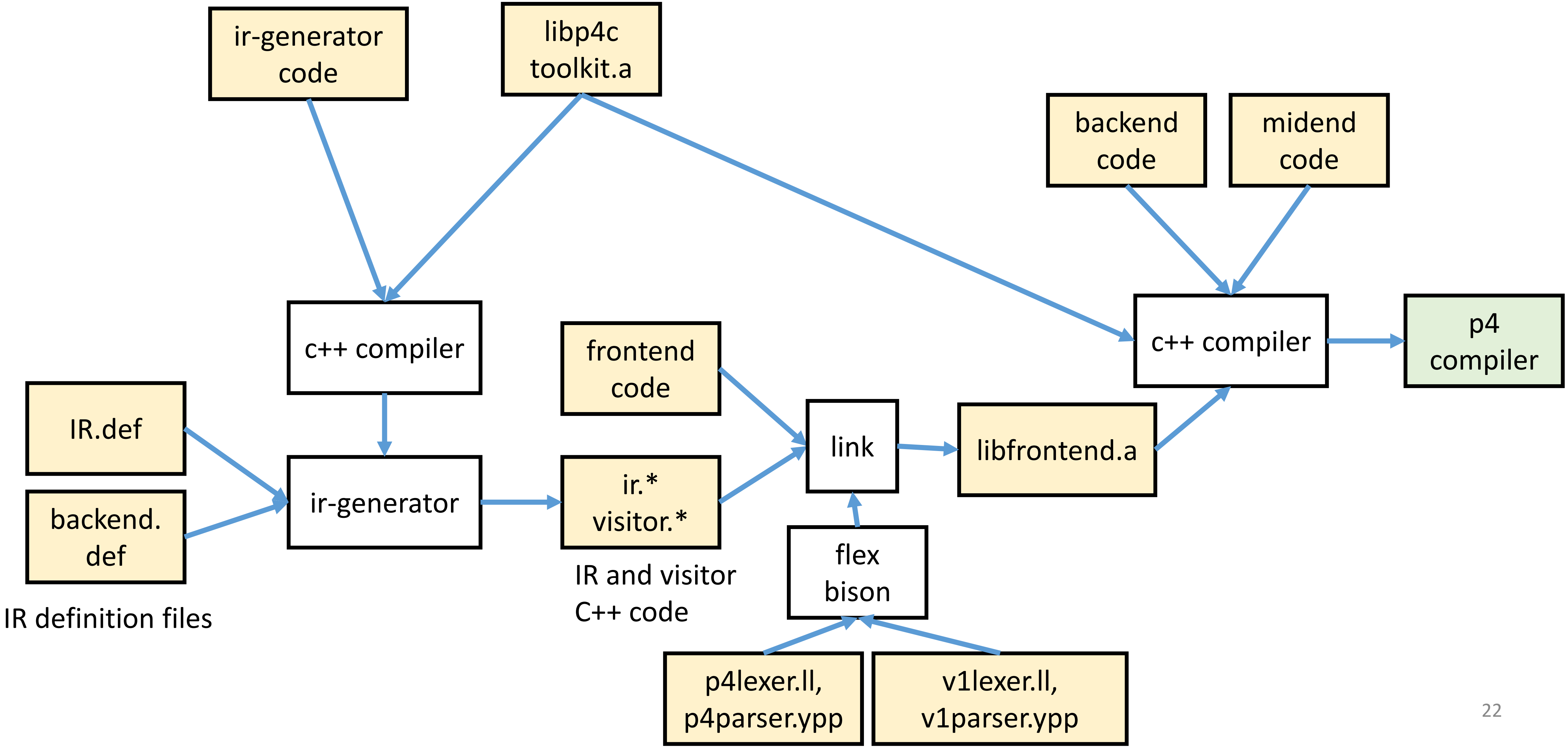
- https://github.com/p4lang/p4c.git
- Required software is described in README.md
  - Need a U*X system (Linux or MacOS)

- To build:

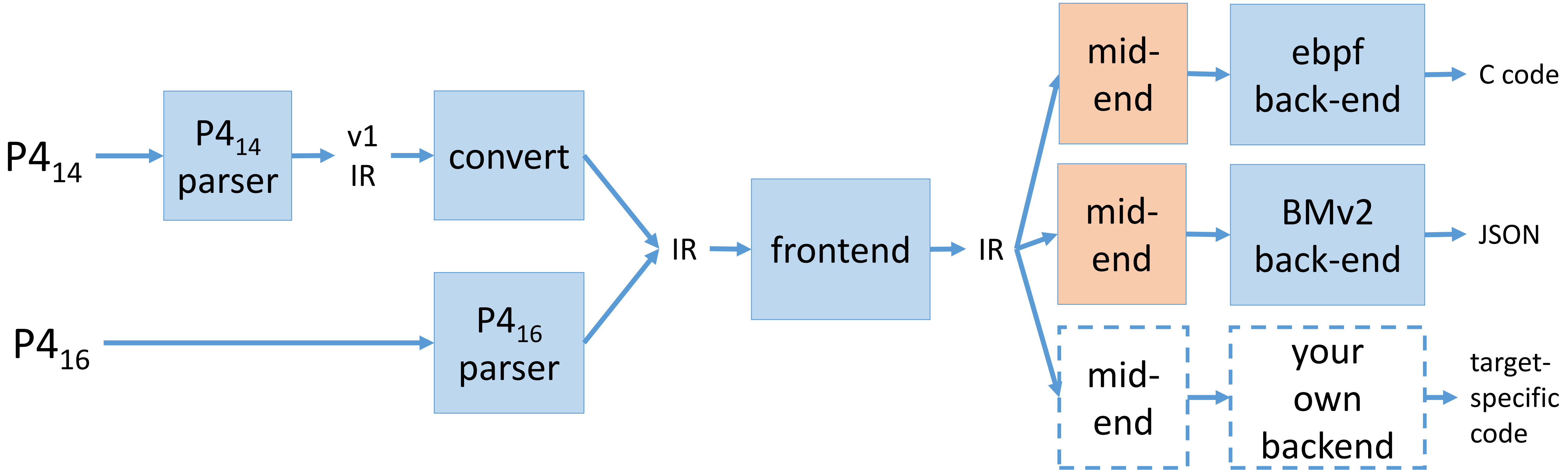
```
cd p4c
./bootstrap.sh
cd build
make –j4
make check –j4
```



Build process



Compiler data flow



Source organization

```
p4c
├── build                    -- recommended place to build binary
├── backends
│   ├── p4test               -- "fake" back-end for testing
│   ├── ebpf                 -- extended Berkeley Packet Filters back-end
│   ├── graphs               -- backend that can draw graphiz graphs of P4 programs
│   └── bmv2                 -- behavioral model version 2 (switch simulator) back-end
├── control-plane            -- control plane API
├── docs                     -- documentation
│   └── doxygen              -- documentation generation support
├── extensions
│   └── XXXX                 -- symlinks to custom back-ends
├── frontends
│   ├── common               -- common front-end code
│   ├── parsers              -- parser and lexer code for P4_14 and P4_16
│   ├── p4-14                -- P4_14 front-end
│   └── p4                   -- P4_16 front-end
├── ir                       -- core internal representation
├── lib                      -- common utilities (libp4toolkit.a)
├── midend                   -- code that may be useful for writing mid-ends
├── p4include                -- standard P4 files needed by the compiler (e.g., core.p4)
├── test                     -- test code
│   └── gtest                -- unit test code written using gtest
├── tools                    -- external programs used in the build/test process
│   ├── driver               -- p4c compiler driver: a script that invokes various compilers
│   ├── stf                  -- Python code to parse STF files (used for testing P4 programs)
│   └── ir-generator         -- code for the IR C++ class hierarchy generator
└── testdata                 -- test inputs and reference outputs
    ├── p4_16_samples        -- P4_16 input test programs
    ├── p4_16_errors         -- P4_16 negative input test programs
    ├── p4_16_samples_outputs -- Expected outputs from P4_16 tests
    ├── p4_16_errors_outputs -- Expected outputs from P4_16 negative tests
    ├── p4_16_bmv_errors     -- P4_16 negative input tests for the bmv2 backend
    ├── v1_1_samples         -- P4 v1.1 sample programs
    ├── p4_14_errors         -- P4_14 negative input test programs
    ├── p4_14_errors_outputs -- Expected outputs from P4_14 negative tests
    ├── p4_14_samples        -- P4_14 input test programs
    ├── p4_14_samples_outputs -- Expected outputs from P4_14 tests
    └── p4_14_errors         -- P4_14 negative input test programs
```

Makefiles

- Using CMake
- The makefiles edited by humans are all called CMakeLists.txt
- There are multiple files, in various folders

Unified builds

- Special trick for compiling C++ programs
- Compiles together many files, and saves times on headers
- Generates a custom `Makefile` from all other `Makefiles`
- Created by `tools/gen-unified-makefile.py`
- You can mostly ignore it

How do I create a new back-end?

- Keep your code in a separate repository
  - Or contribute it to our repository
- Create a symlink to your code in the `extensions` folder
  - e.g., `ln –s myBackEnd extensions`
- Files you have to provide:
  - `CMakeLists.txt` – included in compiler top-level makefile
- You can extend the IR (add new *.def files)

Coding guidelines

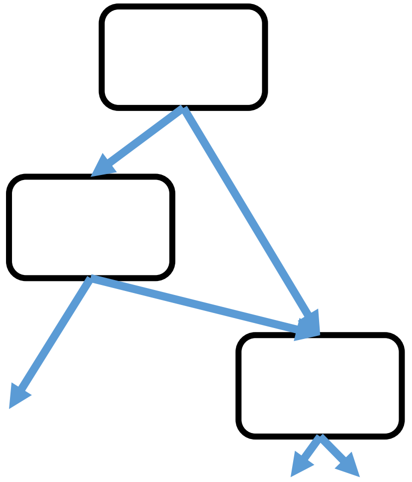
- See files in docs/ folder for coding standards
- Modified Google C++ coding guidelines
- Google's `cpplint.py` with our customized rules (in tools/)
  - `make cpplint` will report all errors
  - `make check` will also invoke cpplint
- To inhibit an error you can use in your code // NOLINT
  - But don't

IR and Visitors

ir/ir.h and ir/visitor.h

Intermediate Representation (IR)

- Immutable
  - Can share IR objects safely
  - Even in a multi-threaded environment
  - You cannot corrupt someone else's state
- Strongly-typed (hard to build incorrect programs)
- DAG structure
  - No parent pointers
  - IR sub-dags can be reused
    - in practice this happens rarely
- Manipulated almost exclusively by visitors
- IR class hierarchy is extensible



shared sub-DAG

IR <=> P4

- Front-end and mid-end maintain invariant that IR is always serializable to a P4 program
- Simplifies debugging and testing
  - Easy to read the IR: just generate and read P4
  - Easy to compare generated IR with reference (testing)
  - Compiler can self-validate (re-compile generated code)
  - Simplifies translation validation (see later)
  - Dumped P4 can contain IR representation as comments
    Use compiler flags `--top4 Pass1,Pass2 -v`
- IR always maintains source-level position
  - can emit nice error message anywhere

Visitor pattern

- https://en.wikipedia.org/wiki/Visitor_pattern

  "In object-oriented programming and software engineering, the visitor design pattern is a way of separating an algorithm from an object structure on which it operates. A practical result of this separation is the ability to add new operations to existing object structures without modifying those structures."

- "Structure" = IR
- "Algorithms" = program manipulations

Visitors

IR classes

Auto-generated

IR manipulations (visitors)

Write only what you need

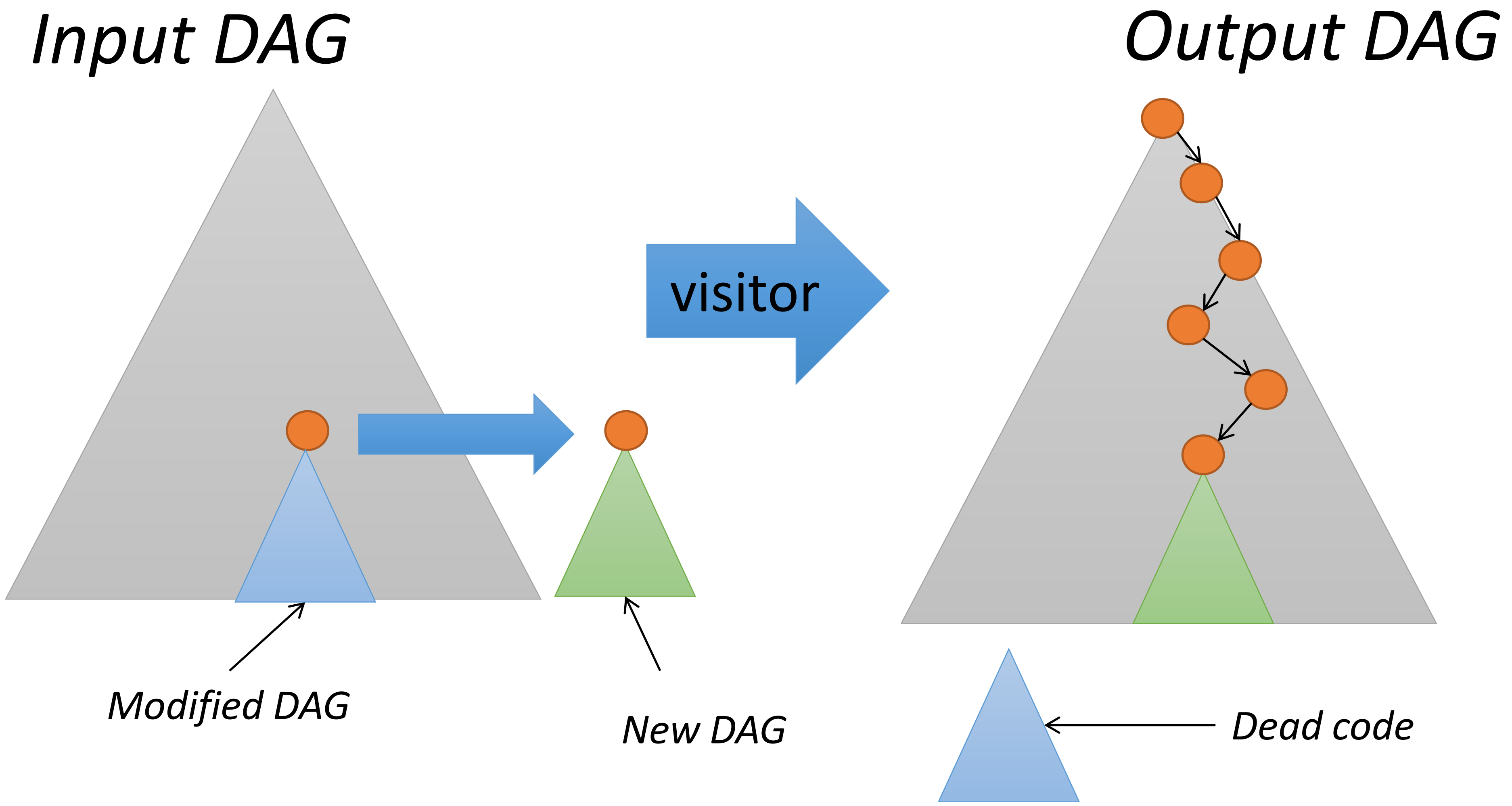
	Add	Subtract	VarDecl	Parser	Control	Header	…
Base	✓	✓	✓	✓	✓	✓	✓
ConstantFolding	✓	✓					
DefUseAnalysis	✓	✓					
DeadCode	✓	✓					
Inlining				✓	✓		
…							

Visitor kinds

See ir/pass_manager.h

Visitor	Description
`Inspector`	Simple read-only visitor that does not modify any IR nodes, just collects information.
`Modifier`	Visitor that does not change the tree/dag structure, but may “modify” nodes in place.
`Transform`	Full transformation visitor.
`PassManager`	Combines several visitors, run in a sequence, manages backtracking.
`PassRepeated`	Repeats a sequence of visitors until convergence.
`VisitFunctor`	Converts a function from Node* to Node* to a visitor.
`PassRepeatUntil`	Repeats passes until a condition is met
`PassIf`	Executes a visitor if a condition is met.

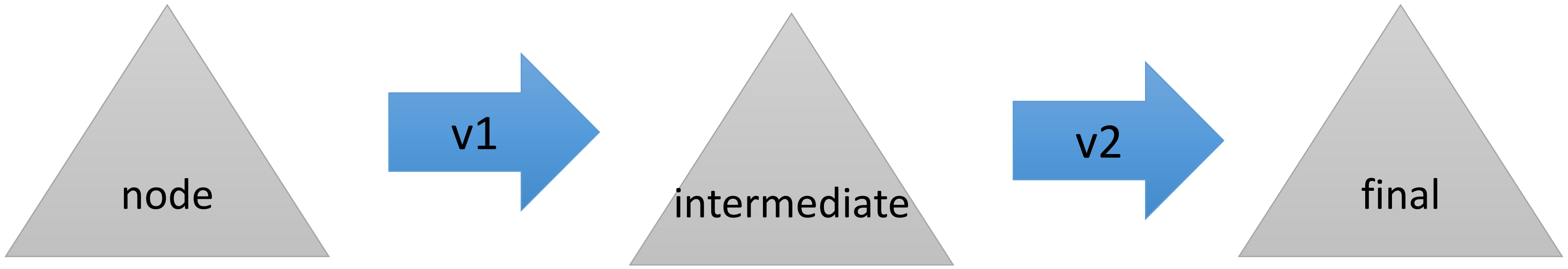
IR rewriting using visitors



Chaining visitors

```
const IR::Node* node;
IR::Visitor v1, v2;

const IR::Node* intermediate = node->apply(v1);
const IR::Node* final = intermediate->apply(v2);
```



IR definition files = Java-like language

```
interface IDeclaration { … }

abstract Expression { … }

abstract Statement : StatOrDecl {}

class AssignmentStatement : Statement {
    Expression left;
    Expression right;
    dbprint{ out << left << " = " << right; }
}
```

Interfaces (pure virtual bases)

Class hierarchy

IR fields

standard IR method

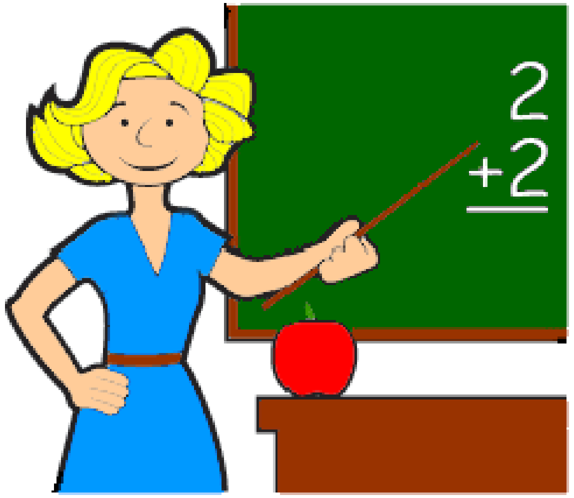
Front-end IR

- ~ 174 concrete classes, 25 abstract classes, 13 interfaces
- $P4_{14}$ (v1.def – 38 classes) and $P4_{16}$ (all other *.def)
- Few classes in common to $P4_{14}$ and $P4_{16}$
- Java-like inheritance
  - INode base virtual class
  - All IR classes descend from Node (node.cpp)
  - Some nodes may implement multiple interfaces
    - e.g., IDeclaration and INamespace
- Core abstract classes
  - Expression – base class for all expressions
  - Type – base class for all types
  - Statement – base class for all statements
  - Declaration – base class for many declarations
  - Type_Declaration – base class for declarations that are also types

Learning the IR by example

Front-end and mid-end passes can all dump IR back as P4 source with IR as comments; use --top4 pass and –v compiler arguments

```
/*
<P4Program>(18274)
  <IndexedVector<Node>>(18275) */
/*
  <Type_Struct>(15)struct Version */
struct Version {
/*
    <StructField>(10)major/0
      <Annotations>(2)
      <Type_Bits>(9)bit<8> */
        bit<8> major;
...
```



IR Generated C++ code (fragment)

- Fields (immutable IR fields)
- debug print
- Equality operator
- Interaction with visitor.
- Invariant checking
- Dynamic type info
- Constructor
- Source position

```
class AssignmentStatement : public Statement {
 public:
   const Expression* left;
   const Expression* right;
   void dbprint(std::ostream &out) const override { out << left << " = " << right; }
   bool operator==(const AssignmentStatement&a) const {
       return Statement::operator==(a)
       && left == a.left
       && right == a.right;
   }
   void visit_children(Visitor &v) override;
   void visit_children(Visitor &v) const override;
   void validate() const override {
       CHECK_NULL(left);
       CHECK_NULL(right); }
   const char *node_type_name() const  { return "AssignmentStatement"; }
   static cstring static_type_name()  { return "AssignmentStatement"; }
   IRNODE_SUBCLASS(AssignmentStatement)
   AssignmentStatement(Util::SourceInfo srcInfo,
       const Expression* left,
       const Expression* right) :
       Statement(srcInfo),
       left(left),
       right(right)
   { validate(); }
};
```

IR Definition language (1)

- C/C++ comments are ignored.
- Subset of C++.
- #emit/#end: enclosed text literally copied to to output .h file
- #emit_impl/#end: enclosed text literally copied to output .cpp file
- #noXXX: do not emit the specified implementation for the XXX method
  - e.g., #noconstructor, #nodbprint, #novisit_children, #nooperator==
- #apply: generate apply overload for visitors
  (rarely needed: makes visitor return same type instead of Node*)

IR Definition Language (2)

- `inline`: Field is not a pointer
- `static`: denotes a static field or method
- `public`, `private`, `protected`, `virtual`, `const`, `namespace`: as in C++
- field initializers
- `optional`: field is not required in constructor
  - Optional field with initializer => can also be set by constructor
- `NullOK`: Field can be a nullptr, otherwise it cannot
- method definition or declaration: as in C++
- *method*{ ... }: specifies an implementation for a default method
  - method can be 'operator=='
- For IR::Operation subclasses some assignments generate methods returning constant values:
  - stringOp: generates cstring getStringOp() const
  - precedence: generates int getPrecedence() const

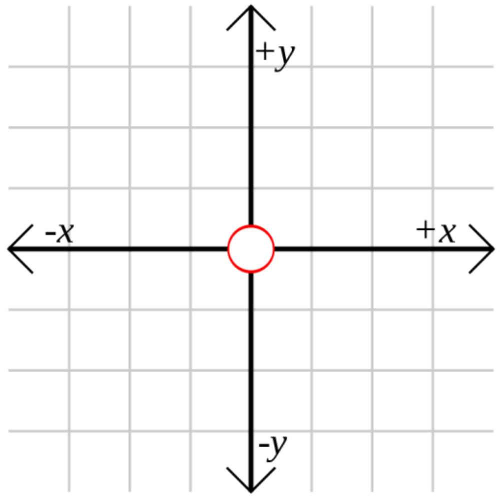
Core IR Methods

- `cstring` **`toString`**`() const` – string representation for compiler user (no internal compiler data structures should be exposed)
- `void` **`dbprint`**`(std::ostream& out) const` – debugging print
- `bool` **`operator==`**`(const N &a) const` – equality comparison performs double-dispatch on this and argument
- `void` **`validate`**`() const` – check construction-time invariants
- `const char*` **`node_type_name`**`() const` – printable class name
- `void` **`visit_children`**`(Visitor &v)` [const] – called by visitor
- `void` **`dump_fields`**`(std::ostream& out) const` – debugging dump
- constructor; arguments inferred from superclasses and fields

Custom hand-coded IR Classes

- `IR::Node` – base of whole class hierarchy
- `IR::Vector<T>` where T is an `IR::Node`
  - `IR::Vector` inherits from `IR::Node`
  - `IR::Vector` stores in fact const T* objects.
  - `IR::Vector` has its own visitor methods
  - Not to be used for other purposes than IR
- `IR::IndexedVector<T>`
  - Like vector, but maintains a hash-table for `IDeclarations` for quick look-up by name
  - Rejects multiple declarations with the same name
- `IR::ID`
  - Represents an identifier (including source position)
  - However, this is **not** a subclass of Node
  - Stores both the original name (provided by user) and the new internal name

Util::SourceInfo



- Represents the source level file position of an IR construct
- Used to provide nice error messages
- When you create new IR nodes consider adding a relevant source position; this will be useful for debuggers and error messages
- Resolving an identifier reference in P4-16 only looks up declarations that are **before** the identifier; it uses the source info for this purpose!
- Default constructor creates an “invalid” source position
  - Invalid source position is logically before all valid source positions

Extending the IR

- Add IR classes in a *.def file
- Add the def file to the CMakeLists.txt:
  - `set(IR_DEF_FILES ${IR_DEF_FILES} *.def PARENT_SCOPE)`
- Add additional c++ IR implementation files to the sources
  - `set(IR_SRCS ${IR_SRCS} ir.cpp)`
- cmake ..; make clean
  - Force regeneration of the IR classes and visitors
- See the bmv2 back-end for a simple example

Casting IR Nodes

- node->is<T>() – true if node is a pointer to a subclass of T
- node->to<T>() – returns node dynamic_cast-ed to const T*
- node->checkedTo<T>() – like to, throws if conversion to T* fails

- interfaces derive from INode, and not from Node
- To get a node from an INode use INode::getNode()

```
const IDeclaration* decl;
const IR::Node* node = decl->getNode();
```

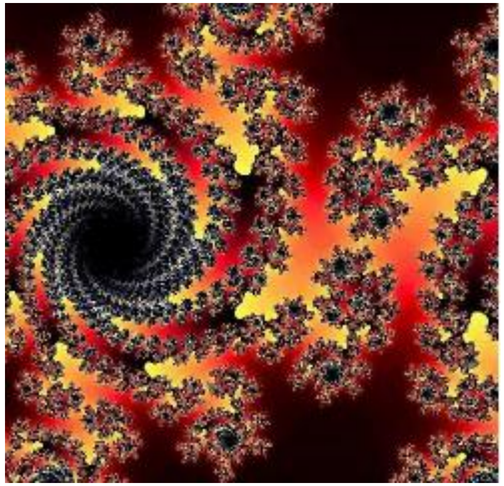
Understanding the front-end IR

- This may seem daunting
- P4 grammar ⬄ IR (very close correspondence)
- If you understand the language, you understand most of the front-end IR
- However, a few IR classes have no direct correspondence with language (e.g., used in representing complex types in type inference)

- E.g., from frontend/parsers/p4/p4parser.ypp:

```
lvalue '=' expression ';' { $$ = new IR::AssignmentStatement(@2, $1, $3); }
```

Visitors and the IR



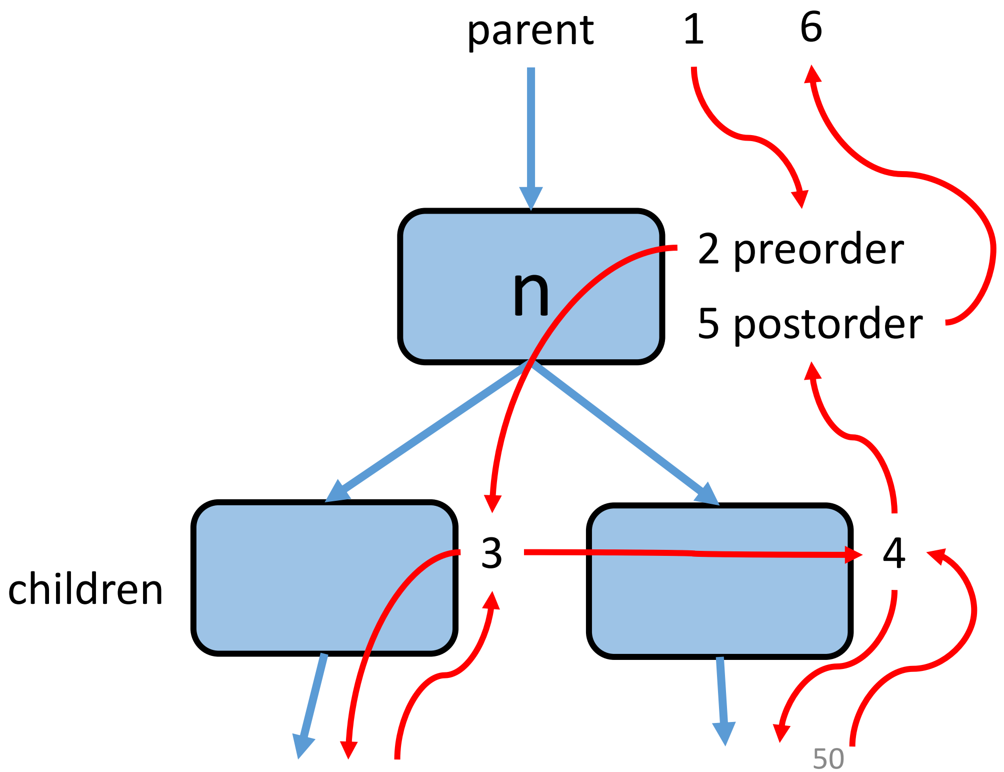
- Tightly coupled
- Visitors recursively traverse the IR children nodes

```
void IR::AssignmentStatement::visit_children(Visitor &v) {
    Statement::visit_children(v);
    v.visit(left);
    v.visit(right);
}
```

Generated code (can be overridden).

Core Inspector action (pseudo-code)

```
const IR::Node *Inspector::apply_visitor(const IR::Node *n) {
     if (visited(n) && visitDagOnce) {
            // do nothing
     } else {
            if (this->preorder(n)) {
                visit_children(n);
                this->postorder(n);
            }
            setVisited(n);
      }
      return n;
}
```



Default implementation

- Visitor base class knows about all IR nodes
- Most of the visitor code is generated automatically by `ir-generator`
- Visitor knows how to create a new node if any child changes
- You will subclass a visitor
- You only need to implement methods for IR node types you care about
  - Everything else works automatically

Example custom visitor declaration

Repeated nodes produce the same result

modifies program

```
class StrengthReduction final : public Transform {
 public:
    StrengthReduction() { visitDagOnce = true; }
    const IR::Node* postorder(IR::Sub* expr) override;
    const IR::Node* postorder(IR::Add* expr) override;
    const IR::Node* postorder(IR::Shl* expr) override;
    const IR::Node* postorder(IR::Shr* expr) override;
    const IR::Node* postorder(IR::Mul* expr) override;
};
```

Types of nodes processed

Example visitor method

```
static bool isZero(const IR::Expression* expr) const {
    auto cst = expr->to<IR::Constant>();
    if (cst == nullptr) return false;
    return cst->value == 0;
}
```

Helper function

```
const IR::Node* StrengthReduction::postorder(
          IR::Add* expr) {
    if (isZero(expr->right)) return expr->left;
    if (isZero(expr->left)) return expr->right;
    return expr;
}
```

Example sequence of passes

```
ReferenceMap refMap;
TypeMap typeMap;

PassManager frontend = {
        new ResolveReferences(&refMap, true),
        new ConstantFolding(&refMap, nullptr),
        new ResolveReferences(&refMap),
        new TypeInference(&refMap, &typeMap),
        new SimplifyControlFlow(&refMap, &typeMap),
        new StrengthReduction(),
};

auto result = program->apply(frontend);
```

Data structures populated by visitors

Pass manager = sequence of visitors

Inspector: builds refMap

Transform: uses refMap

Build refMap for new program

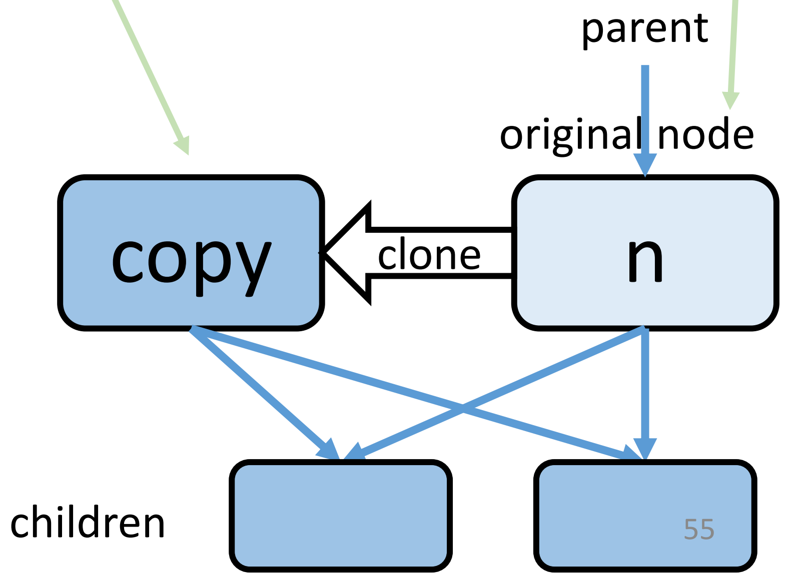
Uses refMap, builds typeMap

Uses refMap and typeMap

Run all visitors in front-end on program in sequence.

Core Transform action (pseudo-code)

```
const IR::Node *Inspector::apply_visitor(const IR::Node *n)
{
    auto copy = n->clone();
    auto preorder_result = preorder(copy);
    if (preorder_result != copy)
        copy = preorder_result->clone();
    copy->visit_children(*this);
    auto final = postorder(copy);
    if (*final != *n)
        n = final;
    return n;
}
```



The original node

- In each visitor method the Node* handed to the method is a *clone* of the original node
- If you store Node* (e.g., in a hash-table) this is a problem
- You can use the `getOriginal()` method to access the original node

```
const IR::Node* SubstitutionVisitor::preorder(IR::Type_Var* tv) {
    auto type = bindings->lookup(getOriginal());
    if (type == nullptr)
        return tv;
    LOG1("Replacing " << getOriginal() << " with " << type);
    return type;
}
```

tv can never be found in the bindings hash table.
We have to index with getOriginal().
tv is actually a temporary clone of the getOriginal() node.

Controlling the visit order

- All visitors visit the children of a node in the order they appear in the visitor class definition
- You can control the visit order by calling visit from preorder.
  - call prune() to inhibit default traversal order (or return false in an Inspector)
- E.g., in an Inspector:

```
bool ToP4::preorder(const IR::StructField* f) {
    visit(f->annotations);
    visit(f->type);
    builder.append(" ");
    builder.append(f->name);
    return false;
}
```

Custom visit order interspersed with side-effects.

Returning ‘false’ causes the current visitor to stop the traversal. This is achieved calling prune() in a Transform.

Transforming and controlling the visit order

- Invoke in `preorder`, call `visit`, and end with `prune`
- Call `prune()` **after** all `visit()` calls only

```
const IR::Node* preorder(IR::If* cond) override {
        auto pred = visit(cond->pred)->to<IR::Expression>();
        ...
        prune();
        return new IR::IfStatement(cond->srcInfo, pred, t, f);
}
```

Custom visit order.

Needs upcast.

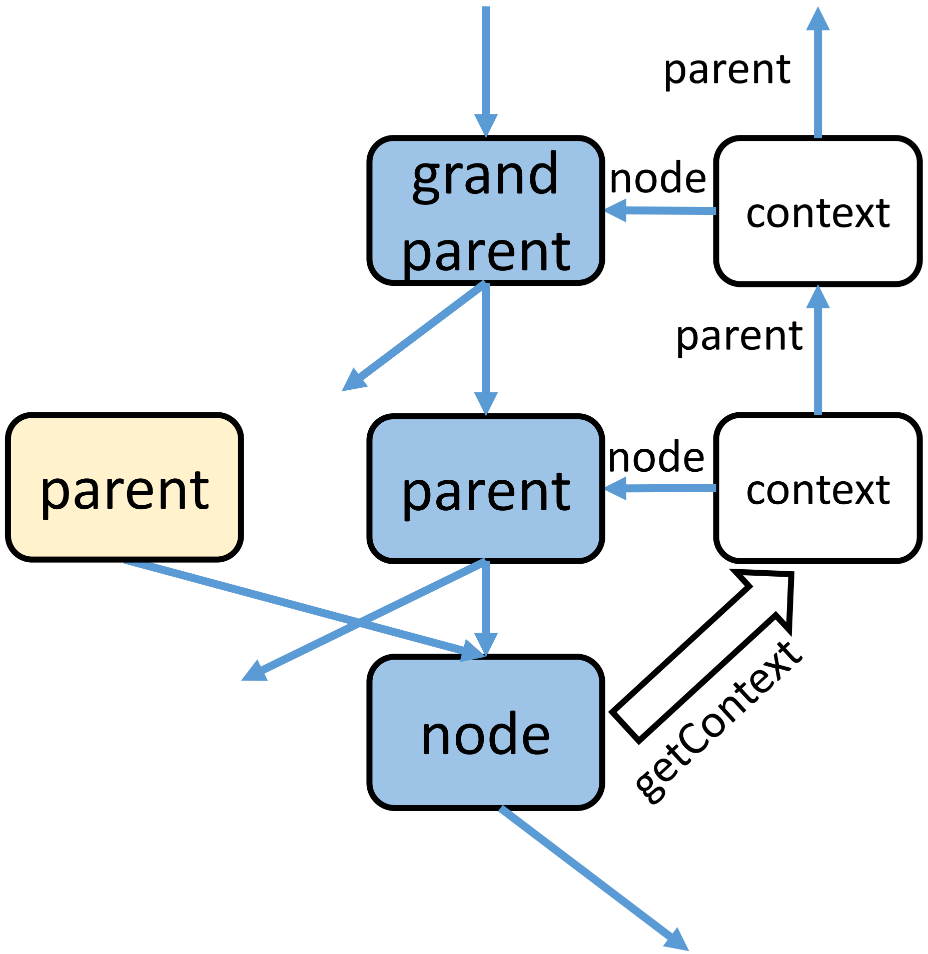
Inhibit standard visit order.

Returns a different class than it is visiting.

Where am I (and how did I get here)

- `getContext()` can tell you how you were reached by the visitor in the IR DAG
- It points to your parent node
- `findContext<T>()` will find an ancestor context for a node of type T

```
const IR::Node* MoveInitializers::postorder(
                IR::Declaration_Variable* decl) {
    if (getContext() == nullptr)
        return decl;
    auto parent = getContext()->node;
    if (!parent->is<IR::P4Control>() &&
        !parent->is<IR::P4Parser>())
        // We are not in the local toplevel declarations
        return decl;
```



Initializing a visitor

- method `init_apply` is called by apply before starting the traversal
- method `end_apply` is called at the end of the traversal
  (but beware that argument Node may have changed between these two calls in a Transform)

```
Visitor::profile_t
TypeChecker::init_apply(const IR::Node* node) {
    LOG2(”Starting type checking”);
    return Transform::init_apply(node);
}
```

Deleting an IR::Node

If the node is part of a parent's `IR::Vector`, `IR::NameMap` or `IR::IndexedVector` you can just return `nullptr`

```
const IR::Node* RemoveUnusedDeclarations::preorder(IR::P4Table* cont) {
    if (!refMap->isUsed(getOriginal())) {
        ::warning("Table %1% is not used; removing", cont);
        LOG3("Removing " << cont);
        cont = nullptr;
    }
    prune();
    return cont;
}
```

Inserting an IR::Node

- If the node is stored in an `IR::Vector<T>` or `IR::IndexedVector<T>`, you can return an `IR::Vector<T>` / `IR::IndexedVector<T>` and it will be spliced within the parent
  - You must use the correct T

```
const IR::Node* SpecializeBlocks::postorder(IR::P4Control* cont) {
    auto insertions = blocks->findInsertions(getOriginal());
    if (insertions == nullptr)
        return cont;

    auto result = new IR::Vector<IR::Node>();
    result->push_back(cont);

    for (auto bs : *insertions) {
        auto newcont = createNewControl();
        result->push_back(newcont);
    }
    return result;
}
```

All P4Control nodes are in a Vector<Node>

Keep original node too

Newly created node to insert after cont

I want to convert the program to something else (e.g. JSON)

- Use an `Inspector`
- Keep a `std::map<const IR::Node*, Util::IJson*> map`;

```
void postorder(const IR::Operation_Binary* expression)
override {
    auto e = new Util::JsonObject();
    e->emplace("op", expression->getStringOp());
    auto l = get(map, expression->left);
    e->emplace("left", l);
    auto r = get(map, expression->right);
    e->emplace("right", r);
    map.emplace(expression, e); // actual result
}
```

Error reporting

- Use `::error()` and `::warning()` for user-induced errors
- These use boost::format format-strings, e.g.,

```
::error("Array indexing %1% applied to non-array type %2%",
         expression, type->toString());
```

- These are smart about handling IR classes and source-level information, e.g.:

```
file.p4(17): error: Array indexing [] applied to non-array type int<2>
        c = a[2];
            ^^^^
```

- They call the `toString()` method on IR classes involved
- One should not expose compiler data structures in error messages

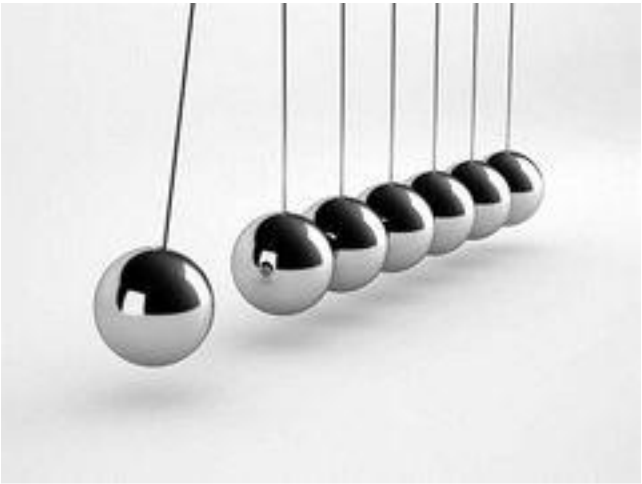
Debugging hints

- To debug the build use `make V=1`
- To debug P4 parsing set YYDEBUG=1 before running the compiler
- To get a stack trace on a compiler crash:
  - (in your back-end you must `setup_signals()` in main (in `lib/crash.h`))
  - run with `–Tcrash:1`
- Use `catch throw` in gdb to break on exceptions
- Set a breakpoint on `::error` in `lib/error.h` to break on errors
- Valgrind is not compatible with the garbage collector library
  - If you want to run the compiler with valgrind disable the GC:
    - `cmake .. –DENABLE_GC=OFF`
    - Of course, you will have lots of leaks

Compiler bugs

- Use the BUG() macro to signal compiler bugs. This macro always throws.
  - Same arguments as for `::error`
  - One ***can*** expose internal data structures when calling BUG
- Don't use assert
- Use CHECK_NULL() to check for null pointers
- Use BUG_CHECK() = assert + BUG in one macro
  - BUG_CHECK(!type->is<IR::Type_Unknown>(), "%1%: Unknown type", f);
- Use P4C_UNIMPLEMENTED to signal a feature not yet implemented (throws)

Determinism



- Keep the compiler deterministic
  - Front-end and mid-ends are all deterministic
- Each node has a unique ID
- (However, clone() preserves the uniqueID!)
- If code is deterministic unique IDs should be reproducible in different runs
- IDs can be used for setting up breakpoints
  - e.g., in Node::trace_creation
- Use `ordered_map` (instead of `std::map`) and `ordered_set` (instead of `std::set`) if you plan to iterate

Dumping IR

::dump(const IR::Node*) dumps the internal IR representation of a node as human-readable text

But using the --top4 –v combination is much easier

Node id

Node type (class)

Fields

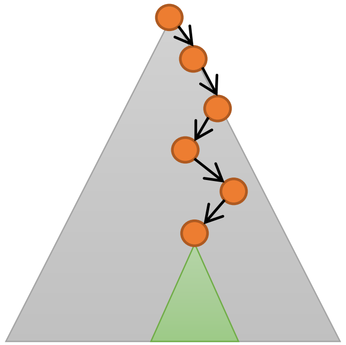
Children are indented

```
[107] P4Program
  declarations: [14] IndexedVector<Node>
    [26] Type_Struct name=P
      annotations: [15] Vector<Annotation>
      fields: [16] IndexedVector<StructField>
        [21] StructField name=f1
          annotations: [17] Vector<Annotation>
          type: [20] Type_Bits size=32 isSigned=0
        [25] StructField name=f2
          annotations: [22] Vector<Annotation>
          type: [24] Type_Bits size=32 isSigned=0
    [37] Type_Struct name=T
      annotations: [27] IndexedVector<Annotation>
      fields: [28] Vector<StructField>
        [32] StructField name=t1
          annotations: [29] Vector<Annotation>
          type: [31] Type_Bits size=32 isSigned=1
        [36] StructField name=t2
          annotations: [33] Vector<Annotation>
          type: [35] Type_Bits size=32 isSigned=1
```

Debugging logs

- Use the LOG*() macros to log internal data structures
  - the LOG macros call the dbprint() method on IR objects
  - `LOG1("Replacing " << id << " with " << newid);`
  - `dbp(const IR::Node*)` is an abbreviated `dbprint`
- Logging is controlled from the command-line with the –T flag:
  - `-Tnode:2,pass_manager:1`
  - logs at level 2 in file node.cpp, and level 1 in pass_manager.cpp
- To specify a header file you must use the full file name
  - E.g., -TinlineCommon.h:3
- E.g., -Tpass_manager:1 will print passes as they are executed

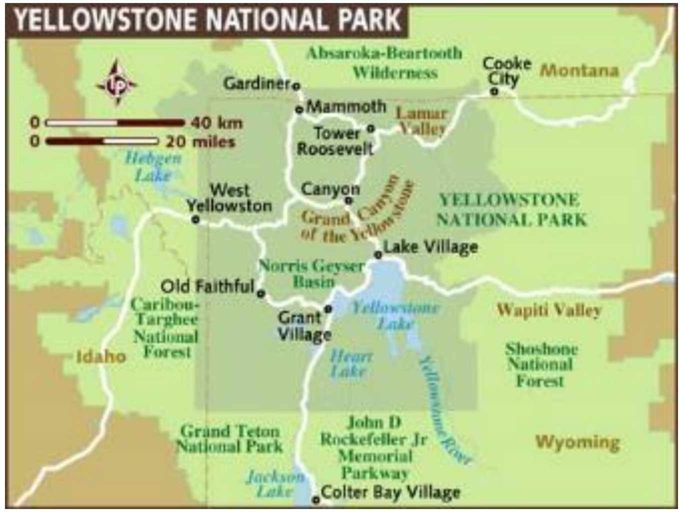
What is the hard part?



- Keeping track of various node versions
- New versions of nodes are created while transformations occur
- Even nodes that you are not touching
  - the ancestors of the nodes you are touching
- References to nodes may become stale
  - pointing to old versions of the nodes, no longer in the IR tree
  - so your carefully constructed maps may need to be reconstructed if you do *anything*
    - e.g., ReferenceMap, TypeMap
- In general, you cannot run two Transforms in sequence if they use some precomputed data structures, since the first will change the program and invalidate the maps

Useful helper classes

- MethodInstance -> applied to a MethodCallExpression, extracts lots of useful information statically
- ConstructorCall -> like MethodInstance, but for ConstructorCallExpressions
- EnumInstance -> helps resolve Enum fields
- ParameterSubstitution -> represents a binding of Expressions to Parameters
  - Use P4::SubstituteParameters to apply a substitution
- P4CoreLibrary -> represents core.p4 library
- TableApply -> helps resolve expressions on tables:
  - table.apply().hit
  - table.apply().action_run
- CallGraph: performs topological sorting, including strongly-connected component computation



A guide to the provided passes

Front-end passes

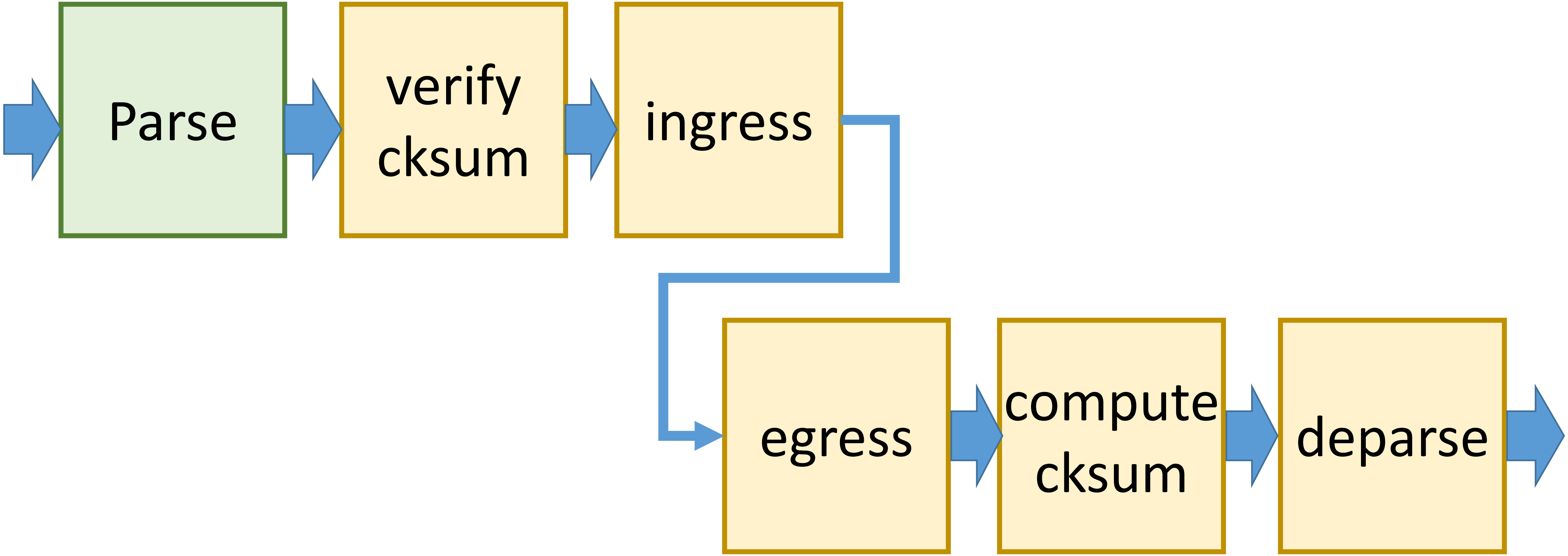
Mid-end passes

$P4_{14}$ (v1.0 / 1.1) front-end

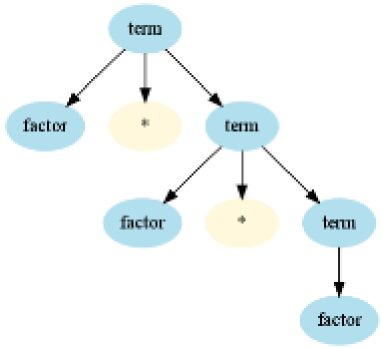
- Code in frontends/p4-14
- Parsed using flex / yacc
- Supports almost all of $P4_{14}$ v1.0 and v1.1
- Some IR classes are only used to represent $P4_{14}$ programs
- Custom $P4_{14}$ type inference
- Converted to $P4_{16}$ IR
- Uses the v1model.p4 architectural model

v1model.p4: A $P4_{14}$ switch model

- A $P4_{16}$ switch architecture that models the fixed switch architecture from the $P4_{14}$ spec
- Provides backward compatibility for $P4_{14}$ programs



Parsing $P4_{16}$



- Parser written using `flex` and `bison`
- Grammar is sometimes difficult to express using bison capabilities
- Parser, lexer and symbol manager cooperate to resolve identifiers
  - Lexer distinguishes types from regular identifiers using symbol table
  - symbol_table.h/cpp

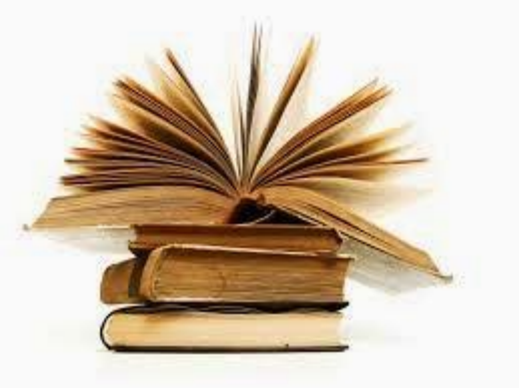
Important P4$_{16}$ classes

- Toplevel element is IR::P4Program
- IR::Constant – integer literal (uses libgmp for arbitrary precision)
- IR::IDeclaration – interface for all classes that introduce a new name
- IR::INamespace – interface for all classes that introduce a new scope
- IR::P4Table – a P4$_{16}$ table ("V1Table" is used for P4$_{14}$)
- IR::P4Parser, IR::P4Control, IR::P4Action – P4$_{16}$ objects
- IR::Type_Control – A control block type declaration (also for Parser, Action, Table)
- IR::Declaration_ID – a declaration that is just an identifier (e.g., in enum)
- IR::Declaration_Instance – instantiates a compile-time object calling a constructor
- IR::Parameter – function/method/block parameter
- IR::Type_Extern – represents an extern block type
- IR::TypeSpecialized – e.g., ext<bit<32>>, where ext is an extern
- IR::TypeNameExpression – e.g., enum X { b } X x = X.b;

Most important passes

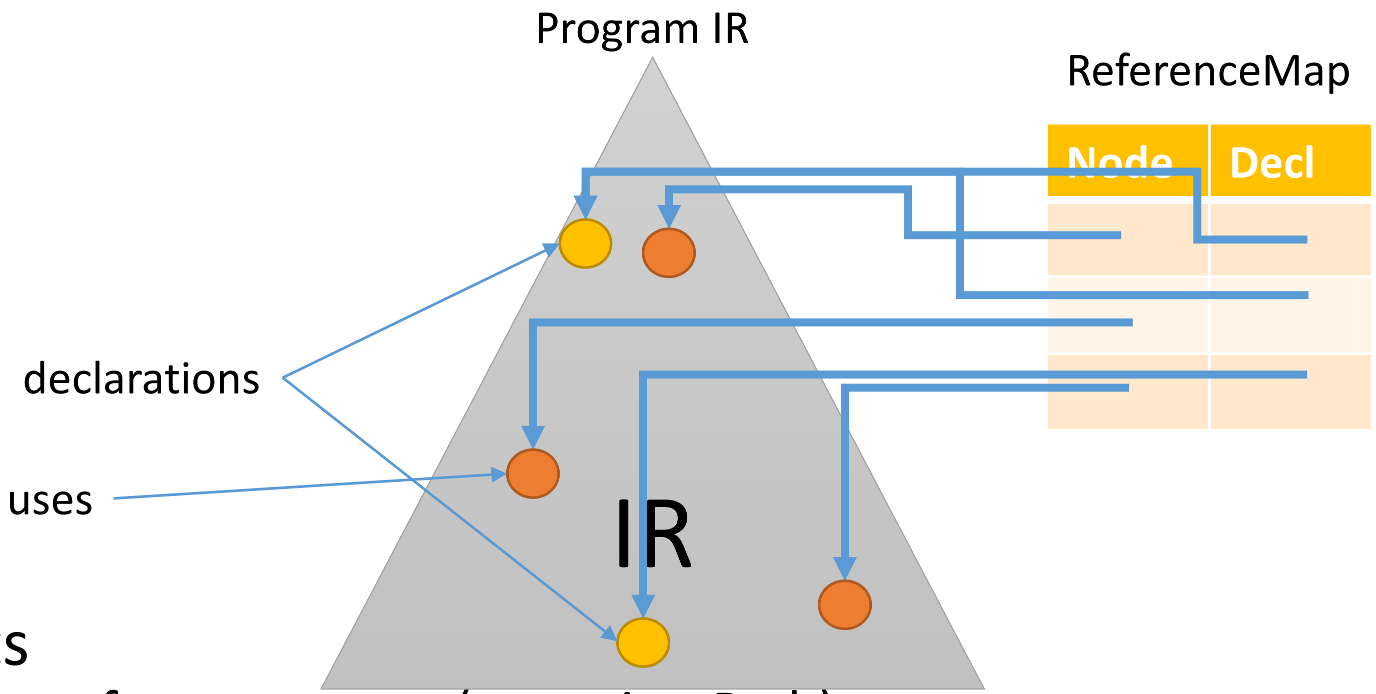
- Need to be rerun every time the program changes
  - ResolveReferences
  - TypeInference
- Evaluator
  - Run after front-end and mid-end
  - Builds the hierarchy of statically allocated resources

ResolveReferences



- Most frequently used pass
  - Called almost every time the program IR changes
- Fills a ReferenceMap
  - Maps each Path to a declaration
  - See below for a description of the ReferenceMap
- (Does not do anything if the program has not changed since the last invocation)
- It must be run starting at the toplevel P4Program
  - Otherwise it may complain about unknown symbols
- Can optionally warn about shadowed symbols
- Scans namespaces inside-out:
  - IR::ISimpleNamespace – at most one declaration with a given name
  - IR::IGeneralNamespace – allows multiple declarations with the same name (e.g., extern methods)

ReferenceMap



- IR::Path generalizes identifiers
- IR::Path can appear in two contexts
  - IR::PathExpression: an expression that refers to a name (name is a Path)
  - IR::Type_Name: an expression that refers to a type by name (name is a Path)
- IR::Member represents a field access
- ReferenceMap core methods:
  - const IR::IDeclaration* getDeclaration(const IR::Path* path)
  - cstring newName(cstring base)
- ResolveReferences fills a ReferenceMap
- Note: source position *is important*: some references are only resolved to previous definitions

INode that introduced symbol referred.

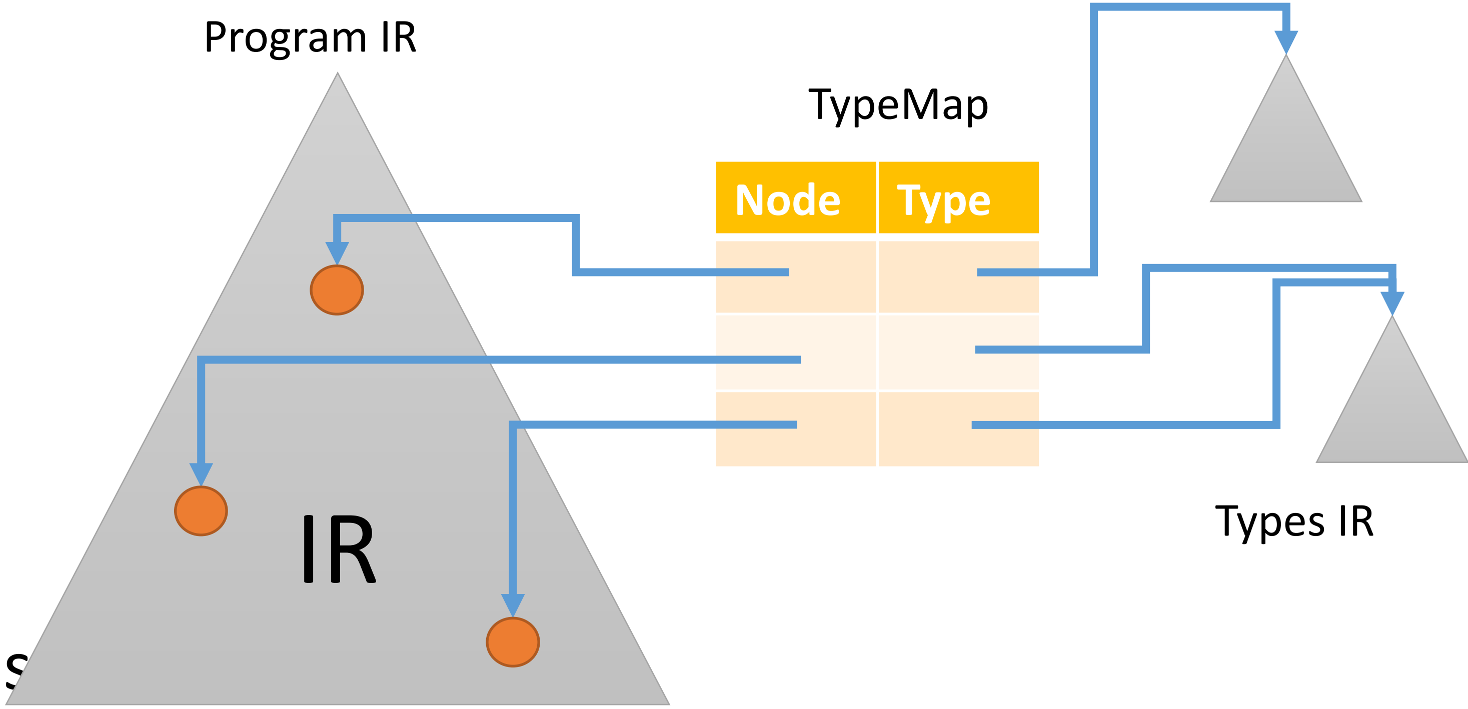
Fresh unique name within the program.

References example

```
const bit<8> x = 10;
struct S { bit<8> s; }
action a(in S w, out bit<8> z)
{
    z = x + w.s;
}
```

```
P4Program
 declarations=Vector<Node>[3]
  0=Declaration_Constant, name=x
    type=Type_Bits, size=8, isSigned=0
   initializer=Constant, value=10
  1=Type_Struct, name=S
    fields=Vector<StructField>[1]
    0=StructField, name=s
      type=Type_Bits, size=8, isSigned=0
  2=P4Action, name=a
   parameters=ParameterList
    parameters=Vector<Parameter>[2]
     0=Parameter, name=w, direction=in
      type=Type_Name
       path=S
     1=Parameter, name=z, direction=out
      type=Type_Bits, size=8, isSigned=0
   body=Vector<StatOrDecl>[1]
    0=AssignmentStatement
     left=PathExpression
      path=z
     right=Add
      left=PathExpression
       path=x
      right=Member
       expr=PathExpression
        path=w
       member=s
```

Type checking



- class TypeInference
- Needs a ReferenceMap
- Checks program typing
- Computes values for type variables
- Inserts explicit casts where needed
- If no casts are needed it should behave like an Inspector and not change the IR
- Produces a TypeMap
  - for each node that has a type the map stores its **canonical** type
  - the canonical representation is not part of the IR program DAG
    (e.g., struct always uses TypeName for fields, but canonical struct has actual field types)
  - not all IR nodes have types

Type checking algorithm

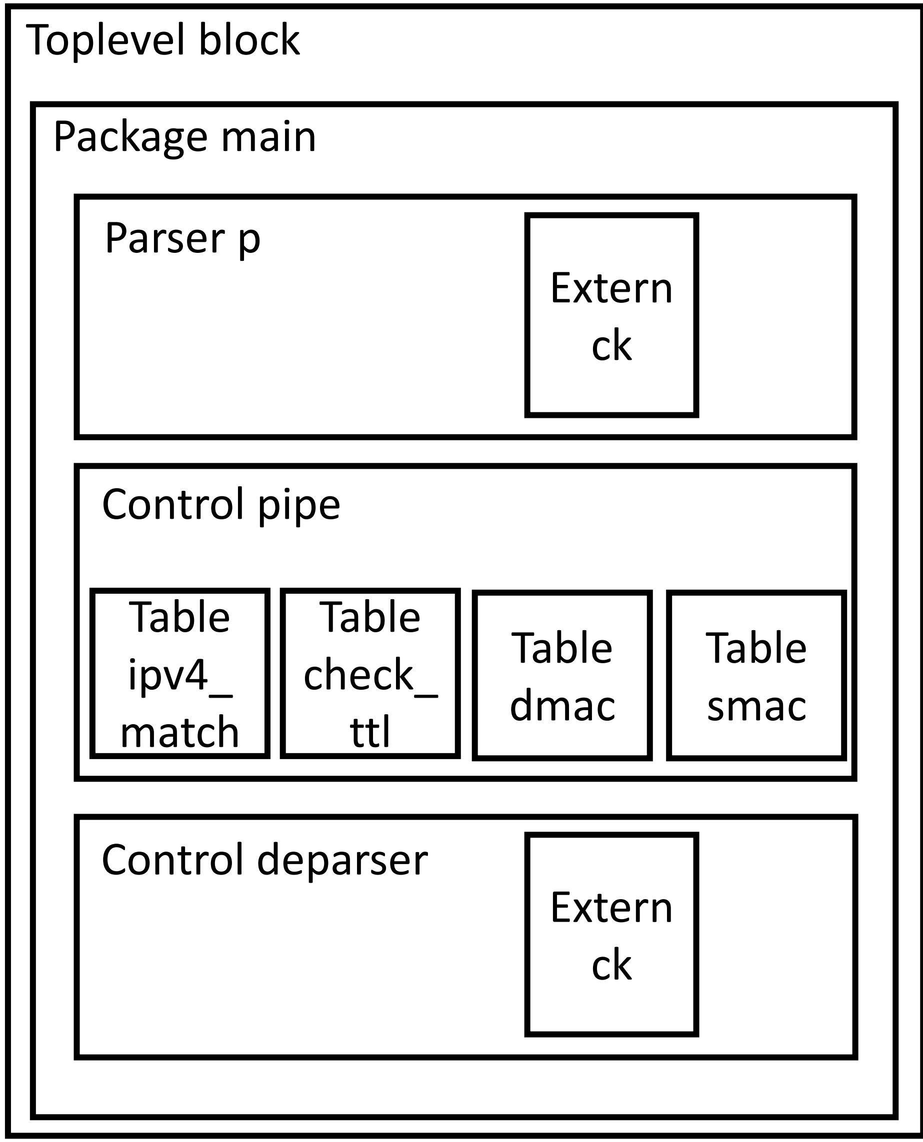
- Somewhat complicated due to generics
- Infers values for unspecified type-variables
- Uses Hindley-Milner (unification) algorithm
  - adapted from http://cs.brown.edu/~sk/Publications/Books/ProgLangs/2007-04-26/plai-2007-04-26.pdf, page 280

- The pass ClearTypeMap erases the typeMap; it should be called when the types of some objects may change (e.g. convert enum to integers)

Evaluation

- P4::Evaluator
- Should be called after the front-end and mid-end
- Represents each program resource as a Block
- Blocks form a DAG
    - children of a block are “allocated” within that block
- Each persistent resource has a block
    - parser, control, packages, externs, tables
- Each block maps IR nodes to CompileTimeValue s
- A CompileTimeValue is a compile-time constant

Block hierarchy for simple-switch-example.p4



The $P4_{16}$ compiler front-end

Front-end passes (frontends/p4/frontend.cpp)

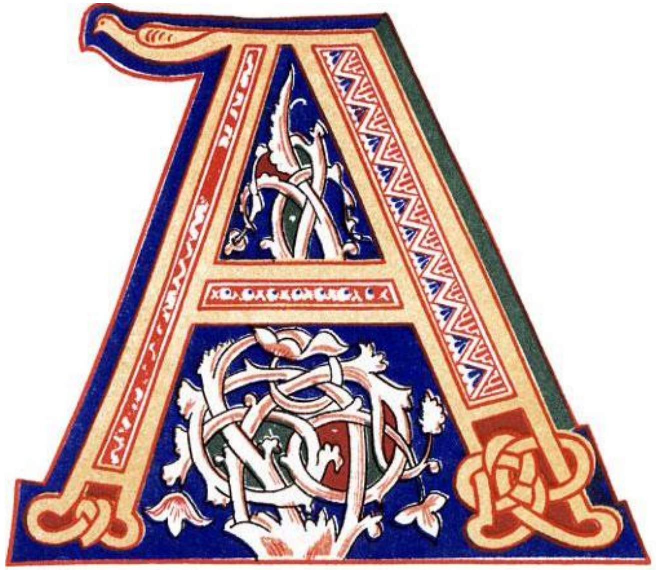
- Pretty printing
- Validation
- Name resolution
- Create control-plane names for keys
- Type checking/type inference (Hindley-Milner)
- Make order of side-effects explicit (argument and short-circuit evaluation)
- Optimizations
- Compile-time evaluation
- Inlining
- Conversion to P4 source

FrontEnd: ParseAnnotationBodies

- Since P4-16 1.2.0 annotations bodies can have free form
  - (anything between a pair of matched parens)
- This pass parses the bodies of annotations that are known to need a specific structure and converts them to IR
- E.g.: @name annotation always expects a string argument

Front-end: PrettyPrint

- Emit program as $P4_{16}$ code
- Used to convert $P4_{14}$ to $P4_{16}$
- Can optionally emit IR as comments in the code
- Enabled with --pp out.p4 compiler flag



Front-end: ValidateParsedProgram

- Run immediately after parsing.
- There is no type information at this point, so it does only simple checks.
  - integer constants have valid types
  - don't care _ is not used as a name for methods, fields, variables, instances
  - width of bit<> types is positive
  - width of int<> types is larger than 1
  - no parser state is named 'accept' or 'reject'
  - constructor parameters are direction-less
  - tables have an **actions** properties
  - table **entries** list are **const**
  - instantiations appear at the top-level only
  - **default** label of a switch occurs last
  - instantiations do not occur in actions
  - constructors are not invoked in actions
  - returns and exits do not appear in parsers
  - exits to not appear in functions
  - **extern** constructor names have proper names
  - names of all parameters are distinct
  - no duplicate declarations in toplevel program

Front-end: CreateBuiltins

- Creates accept and reject states
- Adds parentheses to action invocations in tables:
  - e.g., actions = { a; } becomes actions = { a(); }
- Parser states without selects will transition to reject
- Adds default_action when it is missing; adds NoAction to action list

Front-end: Constant folding

- Can be run before and after type inference
  - More things can be done after types are known
  - E.g., fold casts
- Run several times during compilation
- Run prior to type inference to compute bounds that have to be constant, e.g. e.g., `bit<(3+4)>`
- Also handles some select expressions, detecting some unreachable select labels
- Also handles if statements with constant conditions

FrontEnd: InstantiateDirectCalls

- Converts direct invocations of controls or parsers into separate instantiations and calls
- Convenient syntactic sugar when something is called exactly once

```
control c() { apply {} }
control d() { apply { c.apply(); }}

becomes

control d() {
   @name("c") c() c_inst;
   apply { c_inst.apply(); }}
```

FrontEnd: Deprecated

- Gives warnings if one uses constructs annotated with @deprecated

FrontEnd: CheckNamedArgs

- Checks that named arguments in calls have distinct names
- All arguments must be named or not
- Optional parameters do not have default values

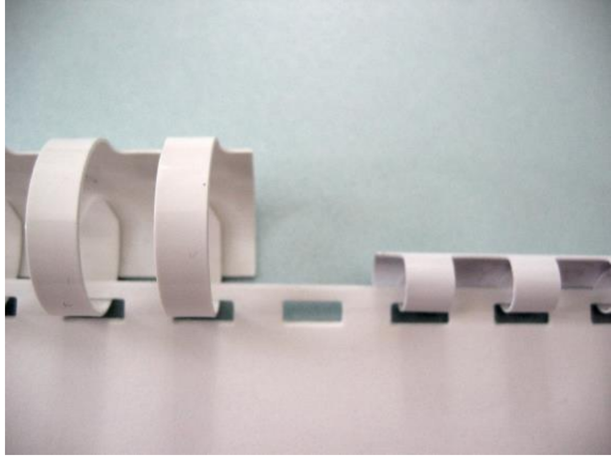
FrontEnd: CheckNamedArgs

- Either all or none of the arguments in a method call may be named.
- No argument appears twice in a call.
- No optional parameter has a default value.

FrontEnd: ValidateMatchAnnotations

- Checks that “match” annotations have a single argument
- Of type match_kind

Front-end: BindTypeVariables



- Type inference should infer values for all type-variables
- This pass replaces type variables with concrete types
  - Constructors, method calls, generic types

```
packet.emit(headers.ipv4);

becomes

packet.emit<IPv4_h>(headers.ipv4);
```

FrontEnd: SpecializeGenericTypes

- Replaces all generic types with a concrete type with the same contents
- For example:

```
struct S<T> { T data; }
S<bit<32>> s;
```

becomes

```
struct S0 { bit<32> data; }
S0 s;
```

FrontEnd: DefaultArguments

- Substitute default arguments when they are not provided
- For example, convert:

```
void f(in bit<32> a = 0);
f();
```

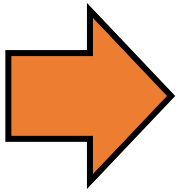
to

```
f(a = 0);
```

FrontEnd: RemoveParserIfs

- Convert an `if` in a parser into a set of new states
- One pass just wraps the other

```
state s {
  statement1;
  statement2;
  if (exp)
    statement3;
  else
    statement4;
  statement5;
  transition selectExpression;
}
```



```
state s {
  statement1;
  statement2;
  transition select(exp) {
    true: s_true;
    false: s_false;
  }
}

state s_true {
  statement3;
  transition s_join;
}

state s_false {
  statement4;
  transition s_join;
}

state s_join {
  statement5;
  transition selectExpression;
}
```

FrontEnd: StructInitializers

- Converts ListExpression to StructExpression where necessary
- StructExpressions have both the type and the field names explicit

FrontEnd: SpecializeGenericFunctions

- Given a function with generic type create a specializ


```
T f<T>(in T data) { return data; }
bit<32> b = f(32w0);
```

Generates the following extra code:

```
bit<32> f_0(in bit<32> data) { return data;
bit<32> b = f_0(32w0);
```

Front-end: TableKeyNames

- Creates a control-plane name for each table key field.
- This enables the compiler to change these expressions later

```
table t { key = { a.x; } … }
```

becomes

```
table t { key = { a.x @name("a.x"); } … }
```

Front-end: StrengthReduction

- Purely syntactic
  - Rewrite div/mod/multiply by powers of two
- Also does some algebraic optimizations
  - add/subtract with 0, shift with zero
  - multiply/divide with 0 or 1
  - bitwise operations with constants
  - DeMorgan laws

Front-end: UselessCasts

- Removes casts where the input and output types are the same

FrontEnd: Reassociation

- Bring together constants in associative operations
- E.g. (a + 2) + 3 is rewritten as a + (2 + 3)
- Facilitates constant folding

Front-End: SimplifyControlFlow

- Remove useless nested block statements
- Simplify if statements with no branches
- Remove empty statements
- Remove unused switch statement labels and empty switch statements
- Removes switch statements with no cases

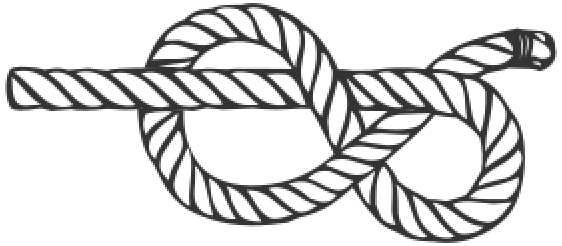
FrontEnd: SwitchAddDefault

- Completes switch statements that do not have all cases covered
- Adds a ‘default: {}’ at the end

Front-End: RemoveAllUnusedDeclarations

- Repeatedly eliminates all declarations that are never referenced in the program
  - control, parser, action, table, variables, parser states
- This is not the same as def-use analysis
- But it does not remove parameters, types, enum members

Front-End: SimplifyParsers



- Remove unreachable parser states
- Collapse straight chains of parser states

Front-End: ResetHeaders

- Inserts code for header.setInvalid() where required
  - Spec indicates that uninitialized headers are invalid

Front-End: SetHeaders

- Headers initialized from lists must also be setValid()
- h = { x }; becomes h.setValid(); h = { x };

Front-end: UniqueNames

- Give each variable in the program a unique new name
- If it is important (e.g., control-plane visible) preserve the old name as a @name annotation.
- Makes it easy to move code around without causing name clashes

Front-end: MoveDeclarations

- Moves all declarations from inner blocks to the outermost scope
- Moves all locals in an action to the enclosing control

Front-end: MoveInitializers

- Variable initialization is separated from declaration
  - In parsers initialization is done in the start state
  - In controls the initialization is done at the beginning of the apply block

```
bit<32> x = 10;
```

becomes:

```
bit<32> x;
x = 10;
```

Front-end: SideEffectOrdering

- Makes evaluation order explicit
- P4 spec mandates left-to-right evaluation order
- Convert expressions such that each expression contains at most one side-effect – by using temporaries and assignments
- Implement short-circuit evaluation for &&, || and ?: converting these expressions into if statements
- Side-effects are caused by function/method calls:
  - Calls may mutate private hidden state (extern/control-plane state)
  - Calls may write to multiple out and inout parameters
- Handles tricky cases such as side-effects in table key computations

FrontEnd: SimplifySwitch

- Constant-fold switch statements that have constant expressions
- These turn into the statement after the corresponding label

Front-end: SimplifyDefUse

- Uses abstract representation of all “locations” (class StorageLocation, class LocationSet)
- Uses abstract representation for “program counter” (class ProgramPoint, class ProgramPoints)
- ComputeWriteSet: computes the locations written at each program point (class Definitions)
  - Inter-procedural analysis for actions and tables
  - Intra-procedural for parsers and controls
- FindUnitialized: finds locations used before being initialized
- RemoveUnused: removes writes to locations that are never read
  - But must preserve method/function side-effects

Front-end: SpecializeAll

- Specialize generic code with constructor parameters for actual types and constructor arguments
- E.g., consider

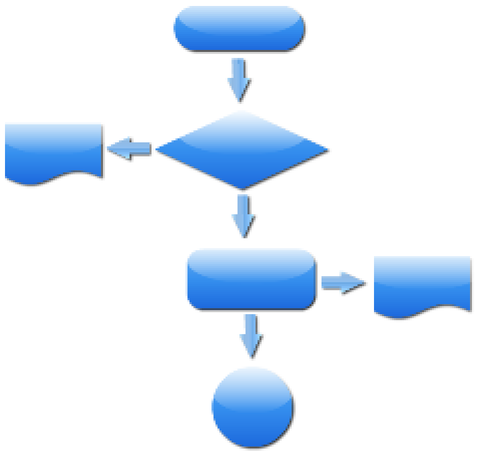
```
control c(out bit<32> o)(bit<32> size)
{ apply { o = size; } }
c(16) c_inst;
```

this is converted to

```
control cspec(out bit<32> o) { apply { o = 16; } }
cspec() c_inst;
```

Front-end: RemoveParserControlFlow

- SideEffectOrdering may introduce `if` statements
- `if` statements are illegal in parser states
- This pass converts such `if` statements into `transition` statements by inserting new states
- Shares code with `RemoveParserIfs`



Front-end: RemoveReturns

- Converts return statements into control-flow
- In actions, functions and control blocks
- In functions there will be exactly 1 return at the end

FrontEnd: RemoveDontcareArgs

- Replaces don’t care arguments with an unused temporary
- This can only happen for ‘out’ parameters

FrontEnd: MoveConstructors

Converts some constructor invocations into instance declarations.

```
extern T { ... }
control c()(T t) {  apply { ... } }
control d() {
     c(T()) cinst;
     apply { ... }}
```

is converted to

```
extern T { ... }
control c()(T t) {  apply { ... } }
control d() {
    T() tmp;
    c(tmp) cinst;
    apply { ... }}
```

Front-end:
Inline, InlineActions, InlineFunctions

- Inline calls to controls from other controls
- Inline calls to parsers from other parsers
- Inline calls to actions from other actions
- Inline calls to all functions (from parsers, controls, functions, actions)
- Inlining requires substituting types, and constructor and call parameters
- Inlining is done bottom-up in the call-graph, starting from leaves
- Inlining creates new hierarchical names for control-plane visible objects (tables, actions)
  - That’s why it is part of the front-end
- One of the most complicated passes in the whole compiler

Front-end: LocalizeAllActions

- Create one action clone for each table using it
- This way actions in different tables can be optimized separately

Front-end: UniqueParameters

- Give unique names to action parameters
- In preparation for parameter removal

Front-end: HierarchicalNames

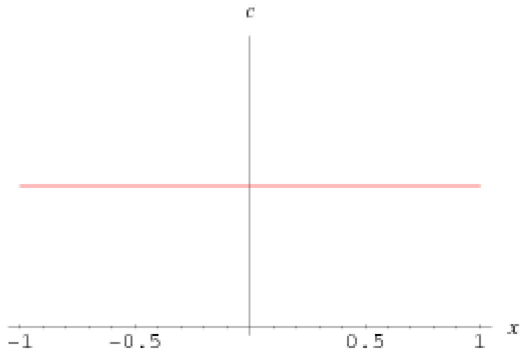
- Gives proper hierarchical names to nested objects

FrontEnd: RemoveActionParameters

- After this pass actions only have control-plane parameters
- The parameters are replaced by variables in the enclosing control

FrontEnd: CheckConstants

- Makes sure that some methods that expect constant arguments have constant arguments (e.g., push_front).
- Checks that table sizes are constant integers.



$P4_{16}$ Compiler Mid-end Passes

Collection of passes that can be assembled by target compiler writers into a custom architecture-specific mid-end

MidEnd: RemoveMiss

- Convert table.apply().miss into !table.apply().hit

Mid-end: SimplifyKey

- Uses a user-supplied policy to decide whether the expression fc computing a table key is too complex
- The key computation can be turned into additional statements

Mid-end: EliminateNewType, EliminateTypedef

- Removes types declared with `type X Y` or `typedef X Y`
- Replaces Y with X everywhere

Mid-end: EliminateSerEnums

- Removes enumerations with a backing type `enum bit<10> E { … }`
- Replaces then with the underlying bit type

(i)
(ii)
(iii)

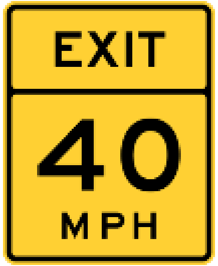
Mid-end: SimplifySelectCases

- If required checks that all select statement labels are constant
- Removes provably unreachable select labels

Mid-end: CompileTimeOperations

- Makes sure that all compile-time only operations have been removed (e.g., division, modulo)

Mid-end: RemoveExits



- Converts exit statements into control-flow
- Inter-procedural: an exit in an action causes the whole control to terminate

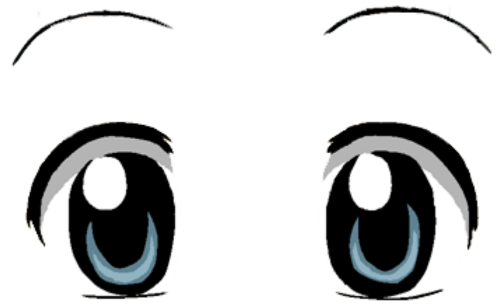
Mid-end: OrderArguments

- Orders calls with named arguments in the order of parameters
- Can be done only if there are no optional parameters

Mid-end: ExpandEmit

- Converts calls to packet_out.emit with arguments that are structures and arrays into multiple calls, one for each field/element

Mid-end: ExpandLookahead

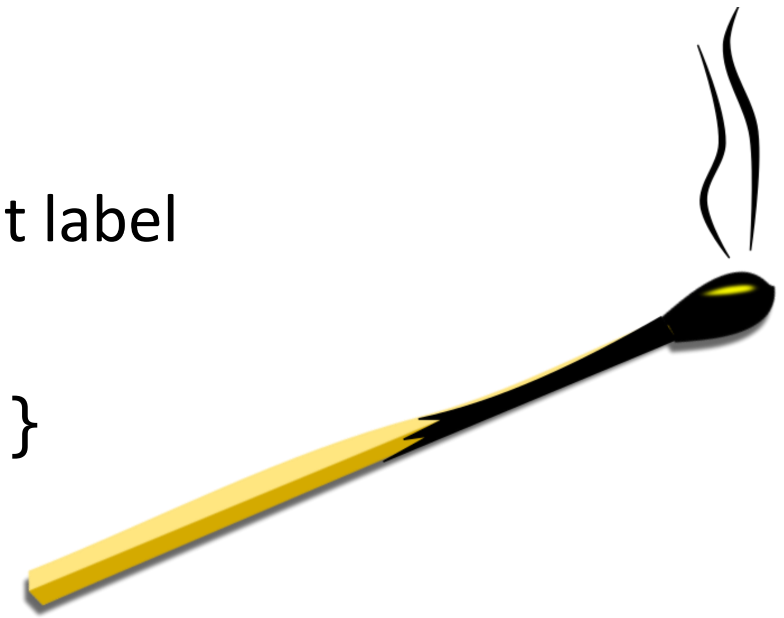


```
struct S { bit<32> f; bit<32> g; }
x = p.lookahead<S>()
```

is converted to:

```
bit<64> tmp = p.lookahead<bit<64>>();
x = { tmp[63,32], tmp[31,0] };
```

MidEnd: HandleNoMatch



- Handles select expressions that do not have a default label

```
state s { transition select (e) { ... } }
```

Is converted into:

```
state s { transition select (e) { ... default: noMatch; }}
state noMatch { verify(false, error.NoMatch);
                transition reject; }
```

Mid-End: EliminateTuples, CopyStructures, NestedStructs, SimplifyComparisons

- Convert tuple<> types to structures
- Convert structure assignments and comparisons to operations between the structure fields (including structure initializers)
- Convert deeply nested structure types to simply-nested structures
  - But it cannot modify parameters to controls or parsers: these are part of the architecture APIs
- In the end structures can only contain scalars, headers or stacks

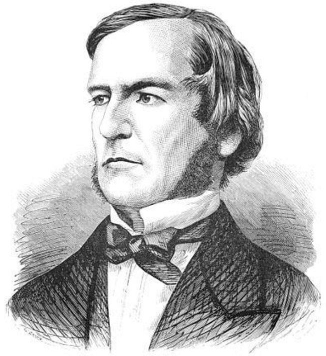
Mid-end: ConvertEnums, FillEnumMaps

- Use a user-supplied policy to convert enum types to bit<> types
- Does not convert enums that are part of the architecture specification
- Preserve enum to value mapping for backend if necessary

Mid-end: LocalCopyPropagation

- Removes some temporary variables

MidEnd: RemoveSelectBooleans



- On targets that do not support Boolean values, this pass can be used to convert all Boolean values that appear in select expressions and labels into bit<1> values

MidEnd: SimplifySelectCases

- If there is just one case label, the select statement is eliminated.
- If a case label appears after the default label, the case is unreachable and therefore eliminated.

MidEnd: SimplifySelectList

- Remove nested types from select expressions

```
transition select(a, b, {c, d}) {
    (0, 0, default): accept;
    (0, 1, {default, default}): accept; }
```

Is converted to:

```
transition select(a, b, c, d) {
    (0, 0, default, default): accept;
    (0, 1, default, default): accept; }
```

MidEnd: FlattenHeaders, FlattenInterfaceStructs

- Converts structs inside headers into lists of fields
- Converts nested structs that are arguments to controls or parsers into flatter types

MidEnd: ReplaceSelectRange

- Converts a select with a range set expression into a sequence of ternary matches

```
(16w0x800, 8w0x8 .. 8w0x10, 8w0x6 &&& 8w0x11): ipv4;
```

- is converted to:

```
(16w0x806, 8w0x8 &&& 8w0xf8, 8w0x8 &&& 8w0xf8): ipv4;
(16w0x806, 8w0x8 &&& 8w0xf8, 8w0x10): ipv4;
(16w0x806, 8w0x10 &&& 8w0xfe, 8w0x8 &&& 8w0xf8): ipv4;
(16w0x806, 8w0x10 &&& 8w0xfe, 8w0x10): ipv4;
(16w0x800, 8w0x8 &&& 8w0xf8, 8w0x6 &&& 8w0x11): ipv4;
(16w0x800, 8w0x10, 8w0x6 &&& 8w0x11): parse_ipv4;
```

MidEnd: Predication

- For targets that do not support conditionals in actions, it converts if statements in actions into ?: statements
- May not always be possible

```
if (e) a = f(b);
```

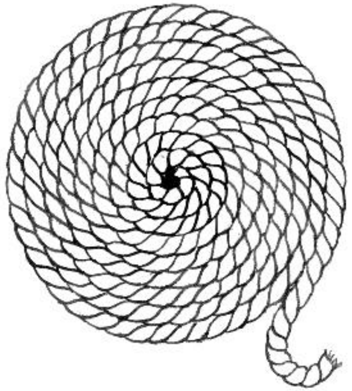
Is converted to:

```
a = e ? f(b) : a;
```

Mid-end: ValidateTableProperties

- Uses a user-supplied policy to checks that that there are no unknown table properties

Mid-end: ParsersUnroll



- Attempts to remove cycles from parser graph
- Based on a symbolic evaluation of the P4 program
- Substitutes the header stacks arguments
- The algorithm could be found at docs/parsersUnroll-readme.md
- In some back-ends triggered by compiler option: --loopsUnroll

Mid-end: ParsersUnroll Simple example

```
…
struct headers {
   …,
   srcRoute_t[2]    srcRoutes;
}
parser MyParser(…, out headers hdr, …) {
state parse_srcRouting {
        packet.extract(hdr.srcRoutes.next);
        transition select(hdr.srcRoutes.last.bos) {
            …
            default:
            parse_srcRouting;
        }
   }
}
```

```
state parse_srcRouting {
        packet.extract(hdr.srcRoutes[0]);
        transition select(hdr.srcRoutes.[0].bos) {
            …
            default:
            parse_srcRouting1;
        }
    }
state parse_srcRouting1 {
        packet.extract(hdr.srcRoutes[1]);
        transition select(hdr.srcRoutes.[1].bos) {
            …
            default:
            parse_srcRouting2;
        }
    }
state parse_srcRouting2 {
      transition stateOutOfBound;
    }
state stateOutOfBound {
     verify(false, error.StackOutOfBounds)
  }
```

Mid-end: SynthesizeActions

- Convert assignment statements in control blocks into actions and action invocations

Mid-end: MoveActionsToTables

- Move all actions that are invoked directly into private tables that have only a default action

MidEnd: RemoveLeftSlices

- Removes slice operations [m,l] on the left-hand side of an assignment

```
a[m:l] = e;
```

Is converted to

```
a = (a & ~mask) | (((cast)e << l) & mask);
```

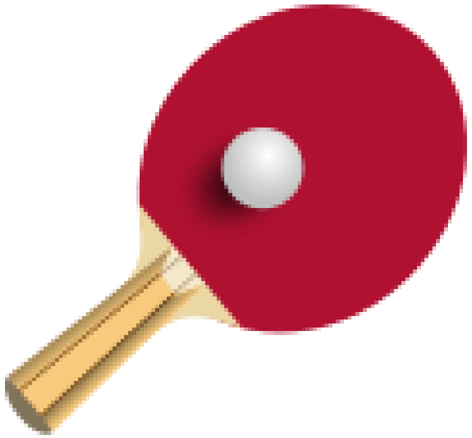
MidEnd: TableHit

- Some architectures can only evaluate table.apply().hit expressions inside conditionals

```
tmp = t.apply().hit
```

Is converted into:

```
if (t.apply().hit)
   tmp = true;
else
   tmp = false;
```



MidEnd: EliminateSwitch

- Converts switch statements that operate on enums or unions into a switch on table applications and actions

MidEnd: ValidateTableProperties

- Makes sure that all properties that appear in tables are known by the current architecture (e.g., implementation)

MidEnd: SimplifyBitwise

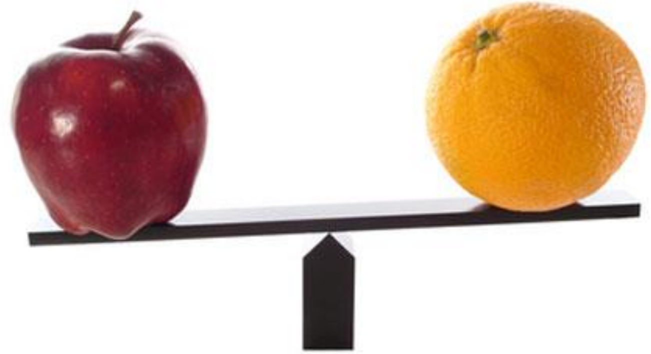
- Optimizes some bitwise patterns (e.g. A & C1 | b & C2) with exclusive masks

MidEnd: RemoveAssertAssume

- If not in debug mode completely delete ‘assert’ and ‘assume’ calls

MidEnd: SingleArgumentSelect

- Convert select(a,b) into select(a ++ b)
- This does not handle don't cares properly, though



MidEnd: ComplexComparisons

- Converts equality comparisons for structs into equality comparisons for all their fields

Low-level IR

- Front-end and mid-end passes:
  - eliminate some IR constructs
  - optimize the IR for a “lower” cost
- Resulting IR is still convertible to P4, but much simpler
  - After front-end:
    - Each declaration has a unique name
    - Each statement has a single “side-effect” (but can write to multiple left-values)
    - All calls can be implemented with copy-in/out or call by reference (no aliasing between arguments)
    - Variables have no initializers
    - All variable declarations are at the top-level scope
    - No type variables exist
    - All integer constants have a known width
    - No constant declarations exist
    - No unused declarations, no unused assignments, no unreachable parser states
    - No divisions, modulo
    - No nested block statements; no empty statements
  - After mid-end (optional, depending on target):
    - Each action is used in only one table
    - No return and exit statements
    - No function, control and parser invocations – all are inlined
    - No parser cycles – all are unrolled
    - No actions called from other actions
    - Actions have only control-plane parameters
    - No nested struct types, no enum types, no tuple types
    - All code in actions; all actions in tables

Sample back-ends

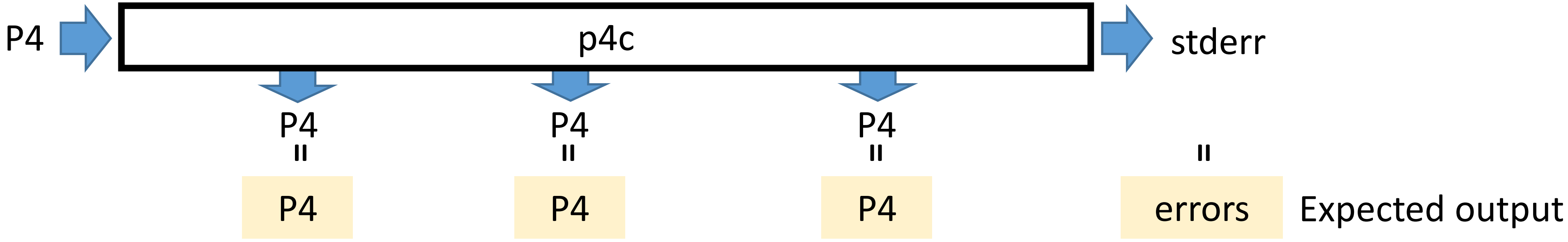
- p4test: back-end used for testing
- p4c-ebpf: P4 => C compiler; C can be compiled to EBPF using BCC or CLANG
- p4c-bm-ss: P4 => JSON compiler; JSON can be loaded by the BMv2 behavioral simulator simple_switch model

p4test

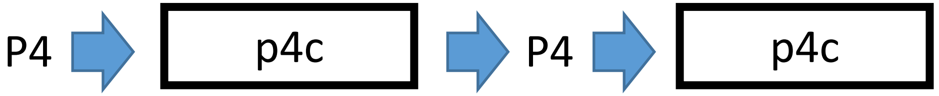
- Fake back-end
- Used for testing the P4 front-ends and mid-ends
- Contains a significant sample mid-end
- Compile files and dump P4 representations
  - Works for both $P4_{14}$ and $P4_{16}$

Testing the compiler

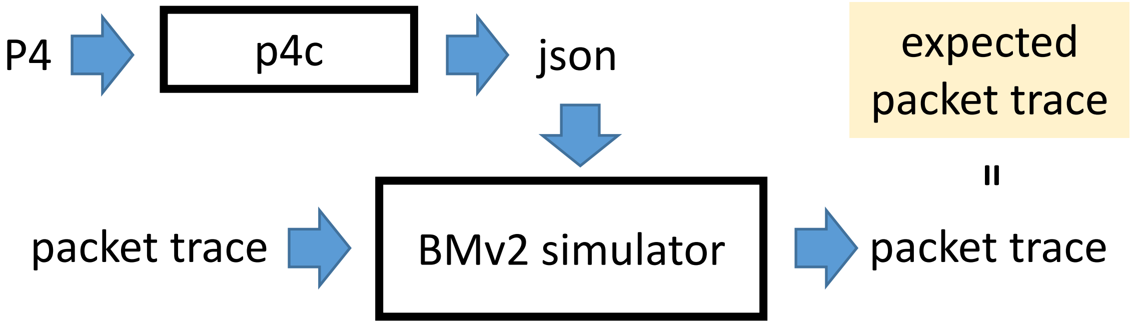
- Dump program at various points and compare with reference
- Compare expected compiler error messages (on incorrect programs)



- Recompile P4 generated by compiler

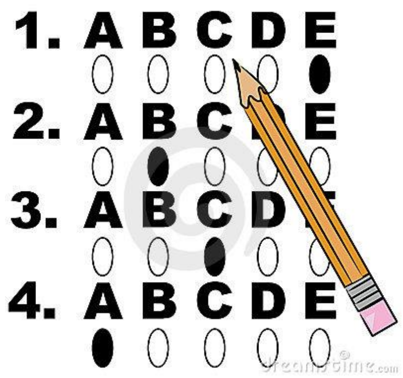


- Run v1model.p4 programs using BMv2 on packet traces and compare to expected output



- Run ebpf_model.p4 programs using C in user-space

Running tests



- make check: runs all tests
- make recheck: runs all tests that have failed last time
- make check-bmv2: run all bmv2 tests
- make check-<pattern>: run tests that match this pattern
- make cpplint: runs the code style checker
- make check PTEST_REPLACE=1: runs tests and replaces all reference outputs. Use with great care, only if you have confirmed that the new reference outputs are all correct. See next slide about replacing individual reference outputs.

Debugging failed tests


```
$ grep ^FAIL: test-suite.log
```

```
FAIL: bmv2/testdata/p4_16_samples/mytest.p4
```

```
$ ./bmv2/testdata/p4_16_samples/mytest.p4.test -v
```

Run test in verbose mode

```
$ ./bmv2/testdata/p4_16_samples/mytest.p4.test -b
Writing temporary files into  ./tmpgI8qqh
...
```

Keep test temporary files

```
$ ./bmv2/testdata/p4_16_samples/mytest.p4.test -f
```

Overwrite test reference outputs

Tests that fail in simulation

- Rerun test to save temporary files

```
$ ./bmv2/testdata/p4_16_samples/mytest-bmv2.p4.test -v -b
Writing temporary files into  ./tmp_cEFKF
Executing  ./p4c-bm2-ss -o bmv2/testdata/p4_16_samples/mytest-bmv2.json
../testdata/p4_16_samples/mytest-bmv2.p4
Exit code  0
Check for  ../testdata/p4_14_samples/bridge1.stf
```

```
$ cd tmp_cEFKF
```

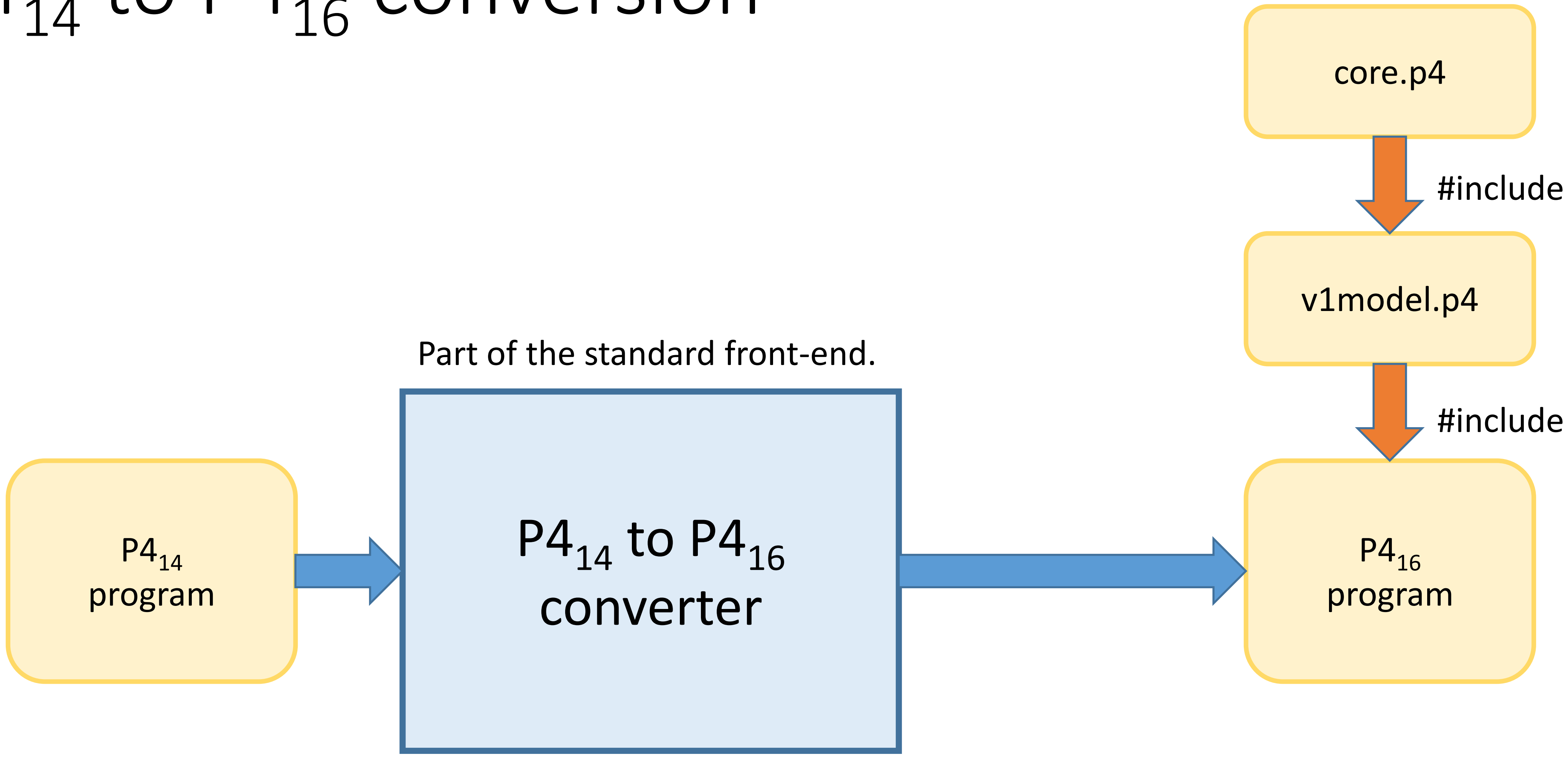
- Rerun the simple_switch simulator manually using the bmv2stf.py script:

```
$ ../../backends/bmv2/bmv2stf.py -v ../bmv2/testdata/p4_16_samples/mytest-bmv2.json \
../../testdata/p4_16_samples/mytest-bmv2.stf
```

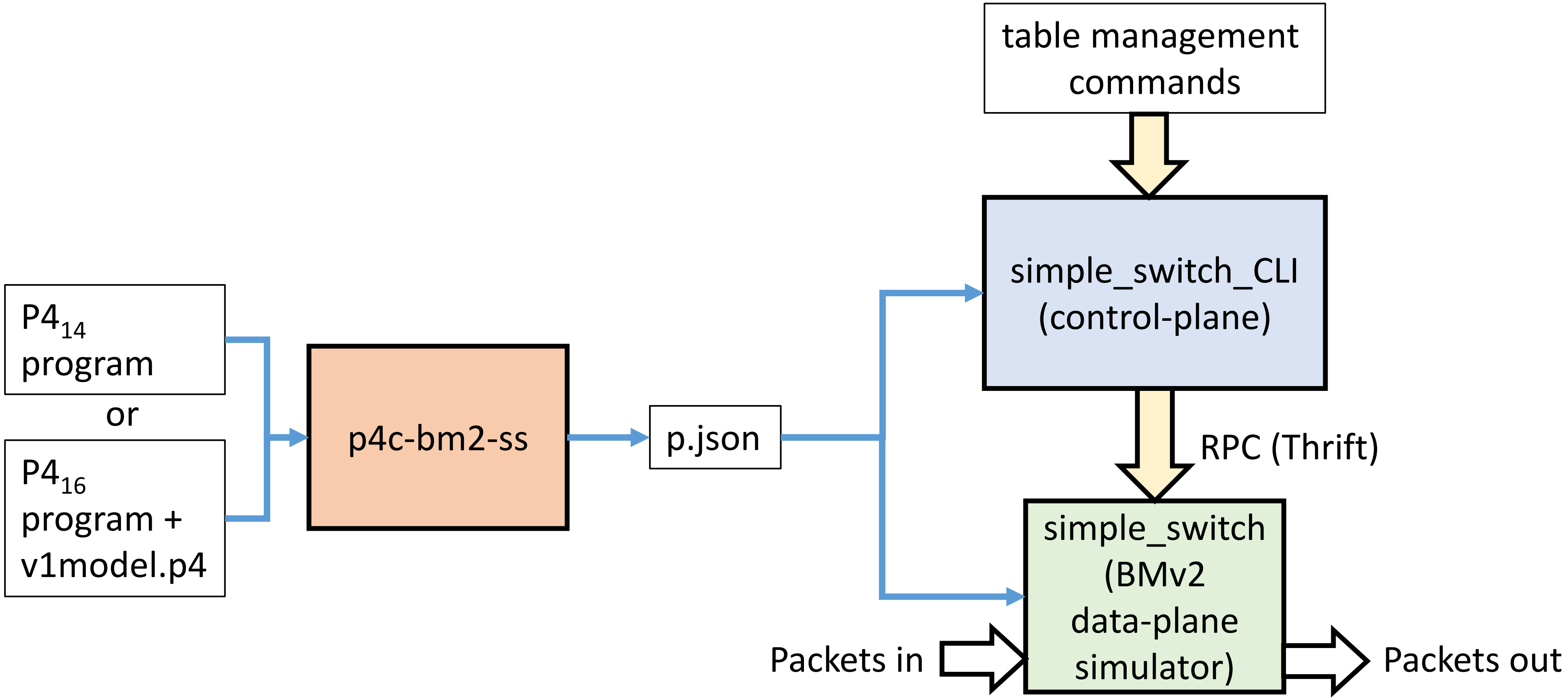
BMv2 back-end

- p4c-bm2-ss
- Target is the software switch `simple_switch` implemented using BMv2 (Behavioral Model version 2) https://github.com/p4lang/behavioral-model
- Handles most $P4_{14}$ programs
  - Converts program to a $P4_{16}$ representation
  - Uses the v1model.p4 architecture
- Can handle simple $P4_{16}$ programs written for the v1model.p4 architecture
- Emits json that can be consumed by `simple_switch`

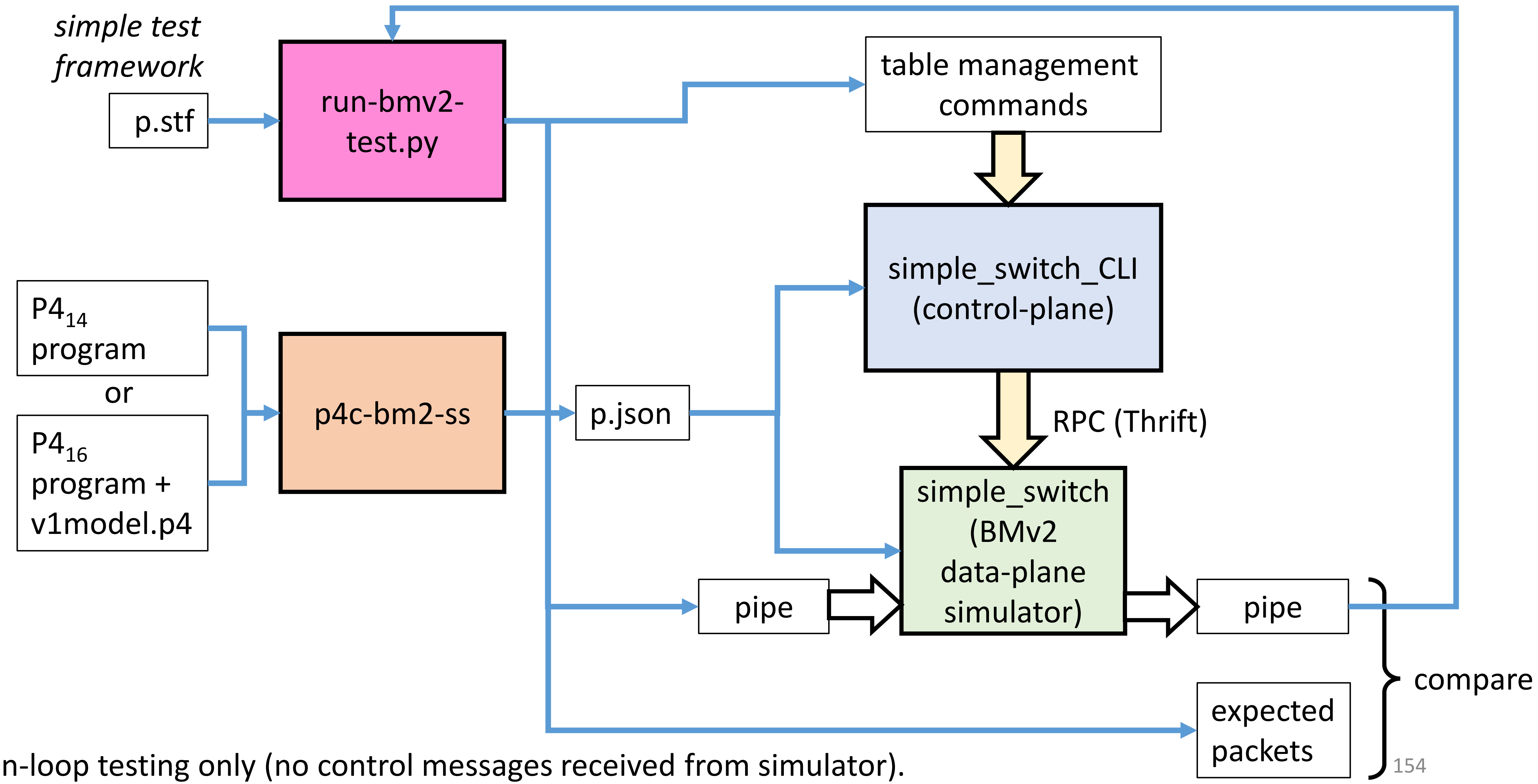
$P4_{14}$ to $P4_{16}$ conversion



Running BMv2



Testing the BMv2 back-end



Currently open-loop testing only (no control messages received from simulator).

Simple test framework language

Table name · Priority · Table key field · Key value (hex, bin or oct) · Action name · Action data

```
add test1 0 data.f1:0x****0101 setb1(val:0x7f, port:2)
add ex1 100 extra$0.h:0x25** act1(val:0x25)
```

Add table entries

```
setdefault test compare()
```

Set default action

```
expect 2 00000101 ******** **** 7f 66
packet 0 00000101 00000202 0303 55 66 7777 88 00
expect 3 00000202 ******** **** 07 66
packet 2 00000202 00000303 0404 55 66 7777 88 00
```

Expected packets

Sent packets

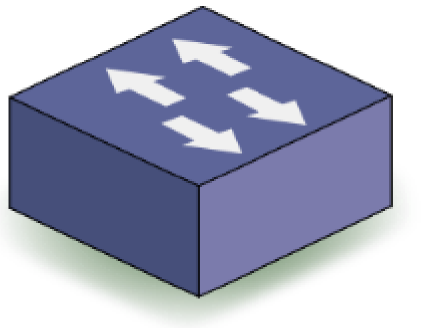
Packet contents in hex (prefix of contents for received packets)

```
# comment
```

Switch interface

* = don't care

Compiling and running switch.p4

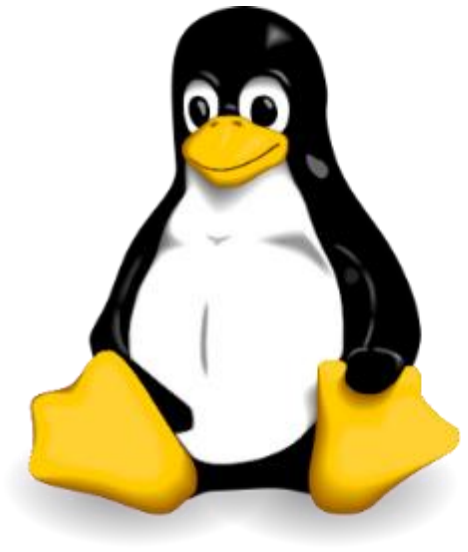


- The largest public P4 program: see https://github.com/p4lang/switch
- Lots of available tests (PTF tests)
- You use the new P4 compiler to generate a JSON file
- You use the old P4 compiler to generate the control-plane APIs from the JSON
- Modify the switch/p4-build/bmv2/Makefile.am as follows:

```
-     PYTHONPATH=$$PYTHONPATH:$(MY_PYTHONPATH) $(P4C_BM) --pd $(builddir)/p4_pd/ --p4-prefix $(P4_PREFIX) --json $(builddir)/$(P4_JSON_OUTPUT) $(P4_PATH)
+       $(P4CNEW) -o $(builddir)/$(P4_JSON_OUTPUT) --p4-14 $(P4_PATH)
+       PYTHONPATH=$$PYTHONPATH:$(MY_PYTHONPATH) $(P4C_BM) --pd-from-json --pd $(builddir)/$(P4_NAME) --p4-prefix $(P4_PREFIX) $(P4C_BM_FLAGS) $(builddir)/$(P4_JSON_OUTPUT)
```

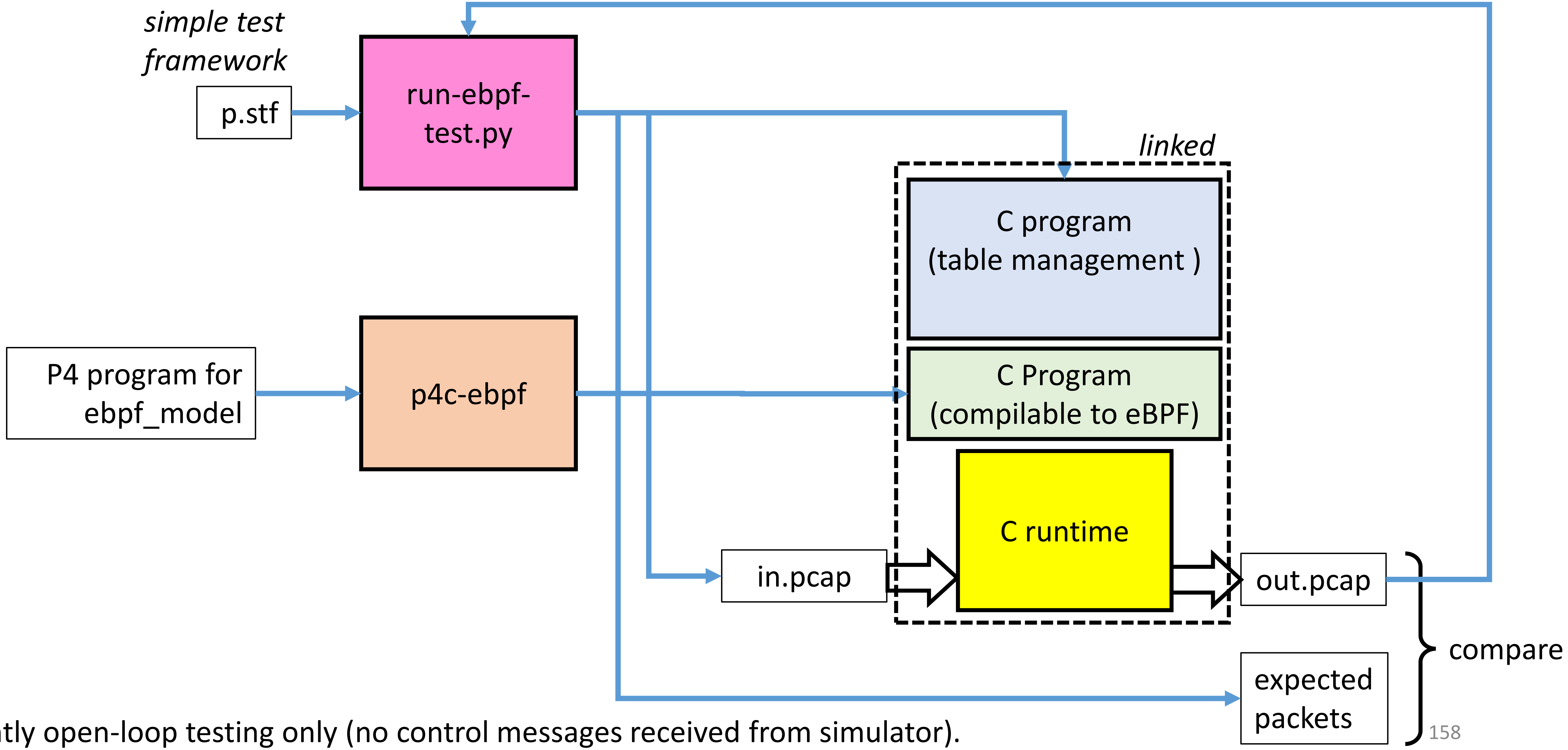
- Make P4CNEW point to p4c-bm2-ss

eBPF Back-end



- https://en.wikipedia.org/wiki/Berkeley_Packet_Filter
- Compiles programs written for ebpf_model.p4
- Converts IR to a restricted subset of C, which can be further compiled using LLVM to eBPF
- Can be used to program the Linux kernel
- Currently restricted to writing packet filters

Testing the eBPF back-end in user-space



Currently open-loop testing only (no control messages received from simulator).

Testing the eBPF back-end in kernel space