DSLs in Compilers

Compiling Techniques

What is a DSL?

- Domain-Specific Language
    - E.g., PyTorch
    - E.g., Matlab
    - E.g., SQL

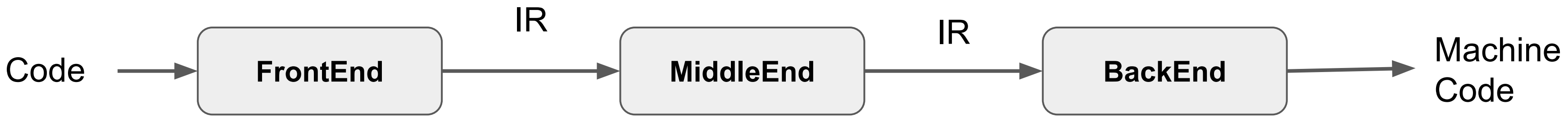
Why do we use DSLs?

- Less Boilerplate
- High-Performance
- Hide Complex Algorithms

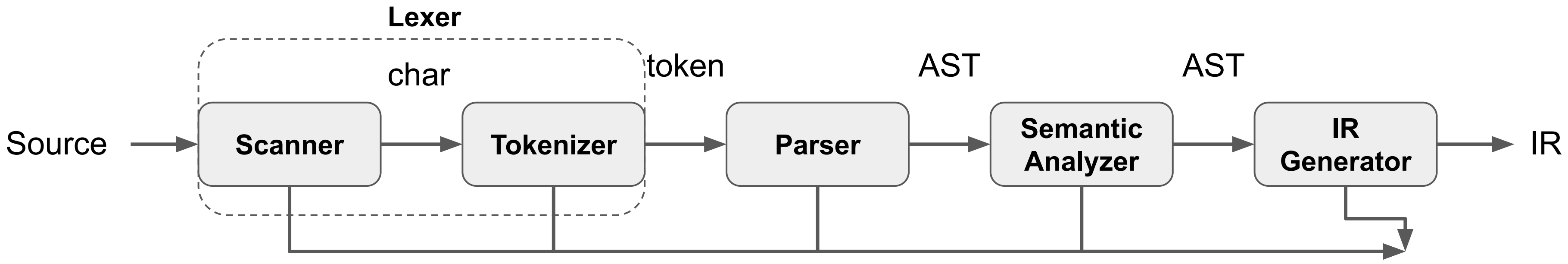
What have the pain points been for you during coursework?

- Lots of boilerplate/simple cases
- Complex cases where it is easy to make a mistake?
- Lots of repetitive tasks?

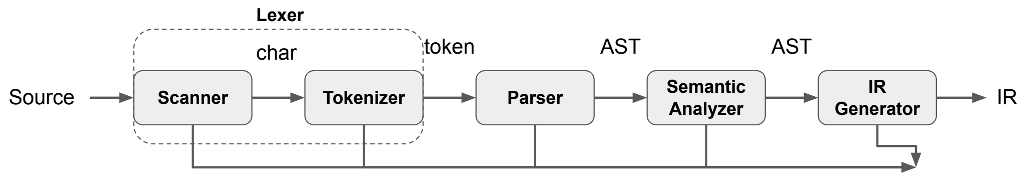
Revisiting Compiler Passes



Reminder from Lecture 1: The Frontend



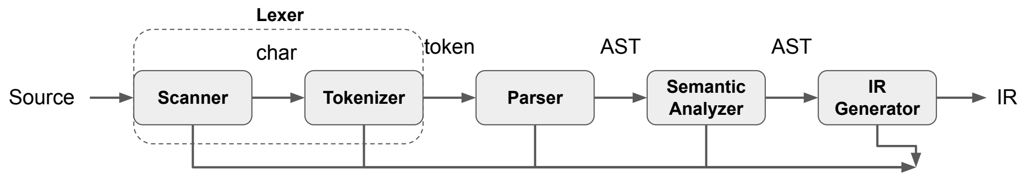
- Recognise legal (& illegal) programs
- Report errors in a useful way
- Produce IR & preliminary storage map
- Shape the code for the back end
- Much of front end construction can be automated



Back to the Start: Lexers

- Lexing task:
  - Identify regular sequences of characters
  - Give them names (the tokens)
- Idea:
  - Use regexes!
  - E.g., for floating point number: `[+-]?([0-9]*[.])?[0-9]+`

Back to the Start: Parsers



- Parsing task:
  - Take tokens and turn into tree
  - Following a specification
- Idea:
  - Use the BNF specification

```
commandline ::= list
          |  list ";"
          |  list "&"

list     ::=  conditional
          |   list ";" conditional
          |   list "&" conditional

conditional ::=  pipeline
          |   conditional "&&" pipeline
          |   conditional "||" pipeline
```

FLEX

Example in C:

```
[ \r\n\t]*    { continue; /* Skip blanks. */ }
[0-9]+        { sscanf(yytext, "%d", &yylval->value); return TOKEN_NUMBER; }

"*"           { return TOKEN STAR; }
"+"           { return TOKEN PLUS; }
"("           { return TOKEN LPAREN; }
")"           { return TOKEN_RPAREN; }

  .             { continue; /* Ignore unexpected characters. */}
```

Bison

```
%token TOKEN_LPAREN   "("
%token TOKEN_RPAREN   ")"
%token TOKEN_PLUS     "+"
%token TOKEN_STAR     "*"
%token <value> TOKEN_NUMBER "number"

%type <expression> expr

/* Precedence (increasing) and associativity:
   a+b+c is (a+b)+c: left associativity
   a+b*c is a+(b*c): the precedence of "*" is higher than that of "+". */
%left "+"
%left "*"

%%

input
    : expr { *expression = $1; }
    ;

expr
    : expr[L] "+" expr[R] { $$ = createOperation( eADD, $L, $R ); }
    | expr[L] "*" expr[R] { $$ = createOperation( eMULTIPLY, $L, $R ); }
    | "(" expr[E] ")"     { $$ = $E; }
    | "number"            { $$ = createNumber($1); }
    ;

  %%
```

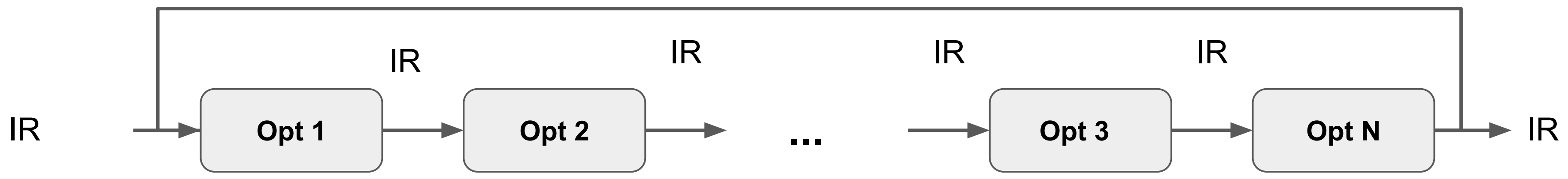
Challenge: Strength of Parser

Modern Approaches: Parser Combinators

```
Expr    ← Sum
Sum     ← Product (('+' / '-') Product)*
Product ← Power (('*' / '/') Power)*
Power   ← Value ('^' Power)?
  Value   ← [0-9]+ / '(' Expr ')'
```

- Combine both into one DSL

Reminder from Lecture 1: The Middle-End



- Discover & propagate some constant value
- Move a computation to a less frequently executed place
- Specialise some computation based on context
- Discover a redundant computation & remove it
- Remove useless or unreachable code
- Encode an idiom in some particularly efficient form

Peephole Optimization

- Rewrite simple patterns
- E.g., X / - 1 => X

Peephole Optimization

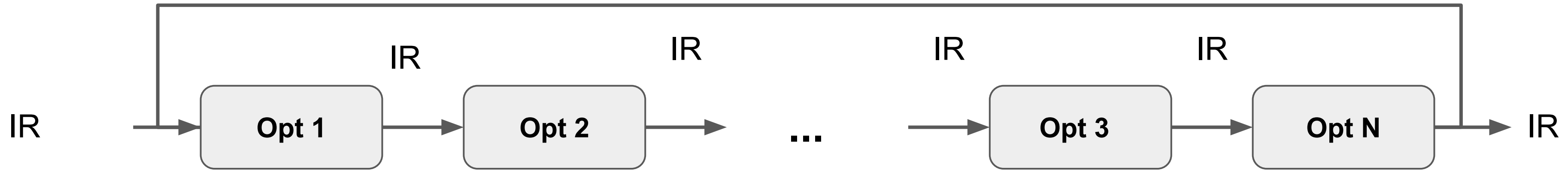
- Idea: Specify as rewrite rules (Pattern Description Language in GCC)

```
/* X / -1 is -X.  */
(simplify
 (div @0 integer_minus_onep@1)
 (if (!TYPE_UNSIGNED (type))
  (negate @0)))
```

Challenges:

- How to tradeoff complex conditions/flexibility with simplicity of expression
- How to build the compiler for pattern description?

Reminder from Lecture 1: The Middle-End



- Discover & propagate some constant value
- Move a computation to a less frequently executed place
- Specialise some computation based on context
- Discover a redundant computation & remove it
- Remove useless or unreachable code
- Encode an idiom in some particularly efficient form

Specifying an IR

- What is an IR?

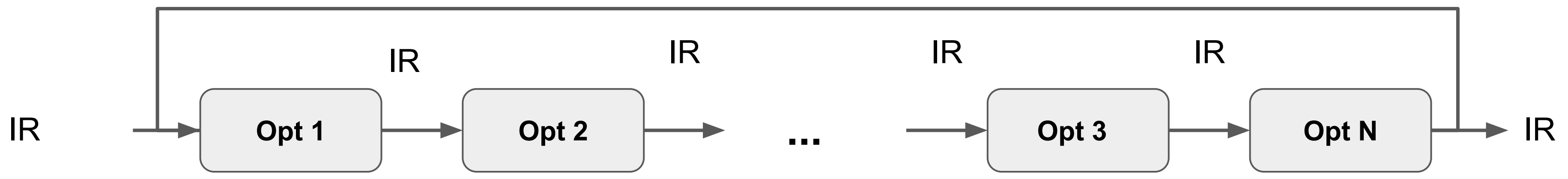
- Idea: Specify operators and arguments and generate helper structures

Specifying an IR - TableGen (MLIR & LLVM)

```
def TF_AvgPoolOp : TF_Op<"AvgPool", [NoMemoryEffect]> {
 let summary = "Performs average pooling on the input.";

 let description = [{
Each entry in `output` is the mean of the corresponding size `ksize`
window in `value`.
 }];

 let arguments = (ins
   TF_FpTensor:$value,
   ConfinedAttr<I64ArrayAttr, [ArrayMinCount<4>]>:$ksize,
   ConfinedAttr<I64ArrayAttr, [ArrayMinCount<4>]>:$strides,
   TF_AnyStrAttrOf<["SAME", "VALID"]>:$padding,
   DefaultValuedAttr<TF_ConvertDataFormatAttr, "NHWC">:$data_format
 );

 let results = (outs
   TF_FpTensor:$output
 );

 TF_DerivedOperandTypeAttr T = TF_DerivedOperandTypeAttr<0>;

}
```

Challenge: Keep complexity without blowing up DSL

Reminder from Lecture 1: The Middle-End



- Discover & propagate some constant value
- Move a computation to a less frequently executed place
- Specialise some computation based on context
- Discover a redundant computation & remove it
- Remove useless or unreachable code
- Encode an idiom in some particularly efficient form

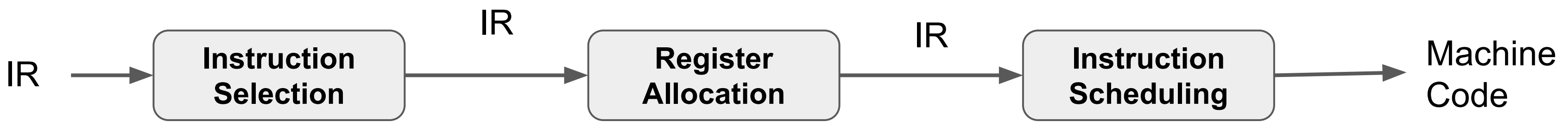
Specifying Optimization Sequences

- What is an optimization sequence?
    - List of transformations
- Idea: List the transformations

Specifying Optimization Sequences: Halide

```
blur_y(x, y) = (blur_x(x, y-1) + blur_x(x, y) + blur_x(x, y+1))/3;
 // The schedule - defines order, locality; implies storage
 blur_y.split(x, xi, yi, 32)
       .vectorize(xi, 8).parallel(y);
```

Reminder from Lecture 1: The Backend



- Translate IR into target machine code
- Choose instructions to implement each IR operation
- Decide which value to keep in registers
- Ensure conformance with system interfaces
- Automation has been less successful in the back end

Instruction Selection

- What did we cover for instruction selection?
    - An algorithm
- Idea: Specify algorithm’s parameters (i.e., the instructions)

Instruction Selection: Machine Description (GCC)

```
(define_insn "*add<mode>3_aarch64"
  [(set
    (match_operand:GPI 0 "register_operand")
    (plus:GPI
     (match_operand:GPI 1 "register_operand")
     (match_operand:GPI 2 "aarch64_pluslong_operand")))]
  ""
  {
    add\t%<w>0, %<w>1, %2
 )
```

Challenges:

- How to manage instruction complexity?
- How to improve performance when you can’t access the algorithm?

Instruction Scheduling

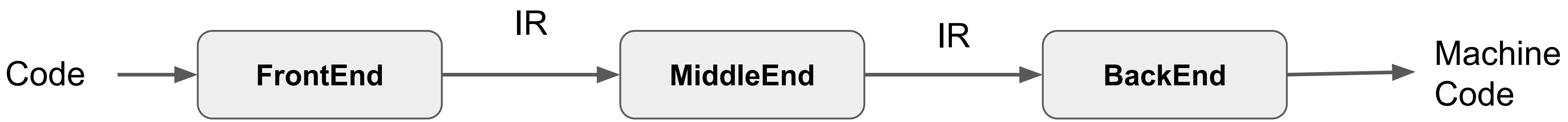
- What did we cover for instruction scheduling?
  - An algorithm
- Idea: Specify Algorithm’s parameters

Instruction Scheduling: TableGen (LLVM)

Describe a CPU architecture at a high level:

```
….

// Divisions.
// These cannot be dual-issued with any instructions.
def : WriteRes<WriteDIV, [M7UnitALU]> {
  let Latency = 7;
  let SingleIssue = 1;
}

// Loads/Stores.
def : WriteRes<WriteLd,    [M7UnitLoad]> { let Latency = 1; }
def : WriteRes<WritePreLd, [M7UnitLoad]> { let Latency = 2; }
def : WriteRes<WriteST,    [M7UnitStore]> { let Latency = 2; }
```

Summary: DSLs at every step of the way



Flex/Bison	Pattern description	Machine description
	TableGen	TableGen
	xDSL	

Conclusion: Look for ways to simplify repeated patterns/complex algorithms in your compiler design — not an exhaustive list!

Overarching Challenges

- How do we manage the complexity that we want for our algorithms, with the simplicity in the DSLs?
- How do we go about improving performance if we can’t access the algorithms?

Summary Summary: Compiling Techniques

Thank you for attending!

You should have:

- A good idea of how compilers work
- Ability to implement a compiler, and to tackle compiler problems you may find in your own work
- The skills to identify how your code changes are modifying code and understand optimizations