Compiler Construction

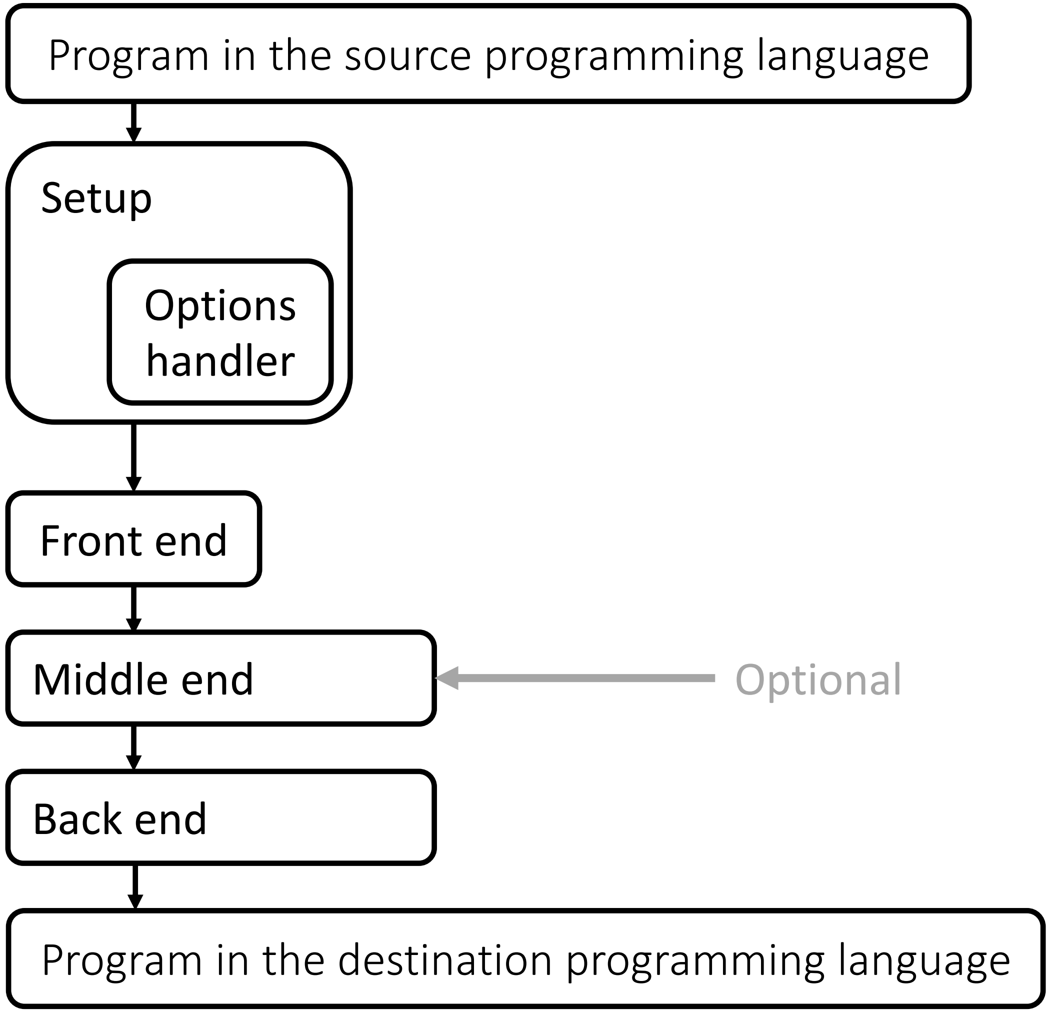
Parsing

Simone Campanoni
simone.campanoni@northwestern.edu

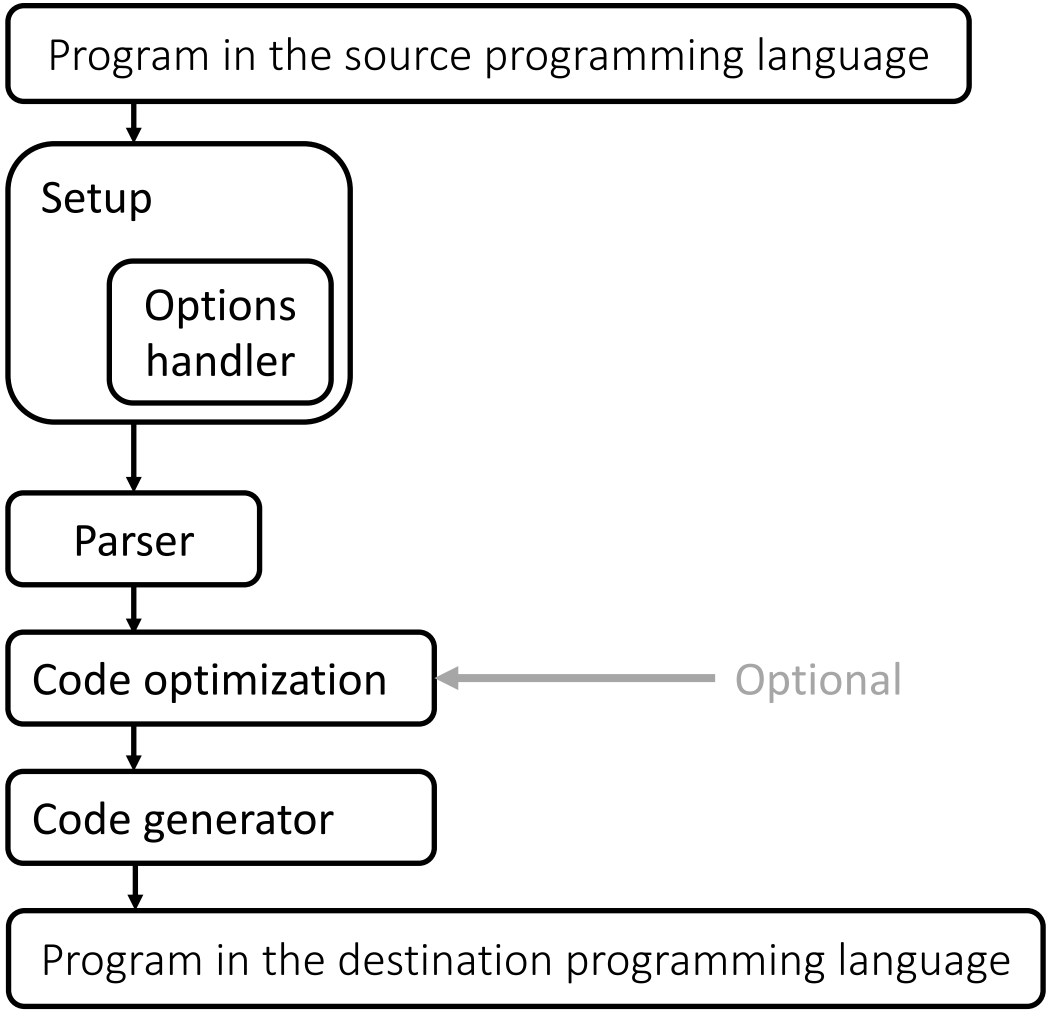
Outline

- Compiler structure
- Parsing
- Parsing with PEG

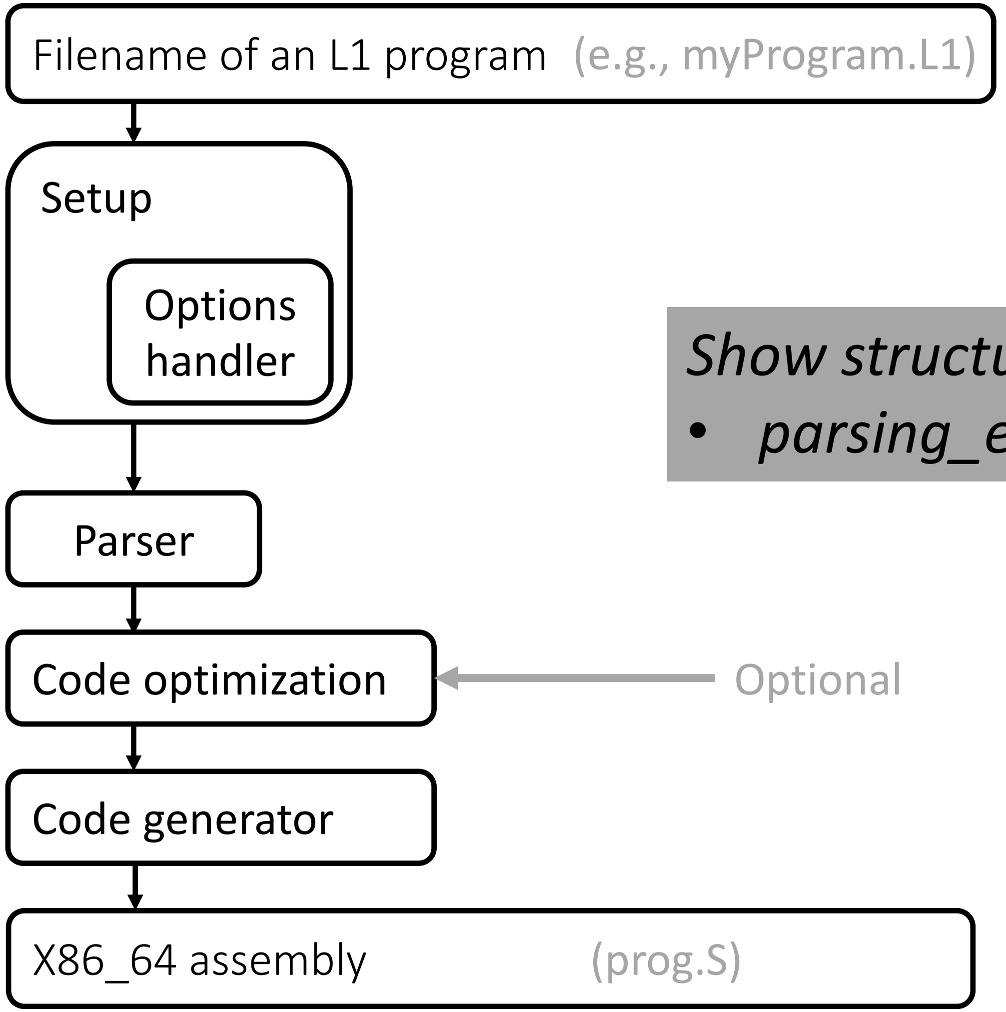
Compiler structure



Compiler structure for this class



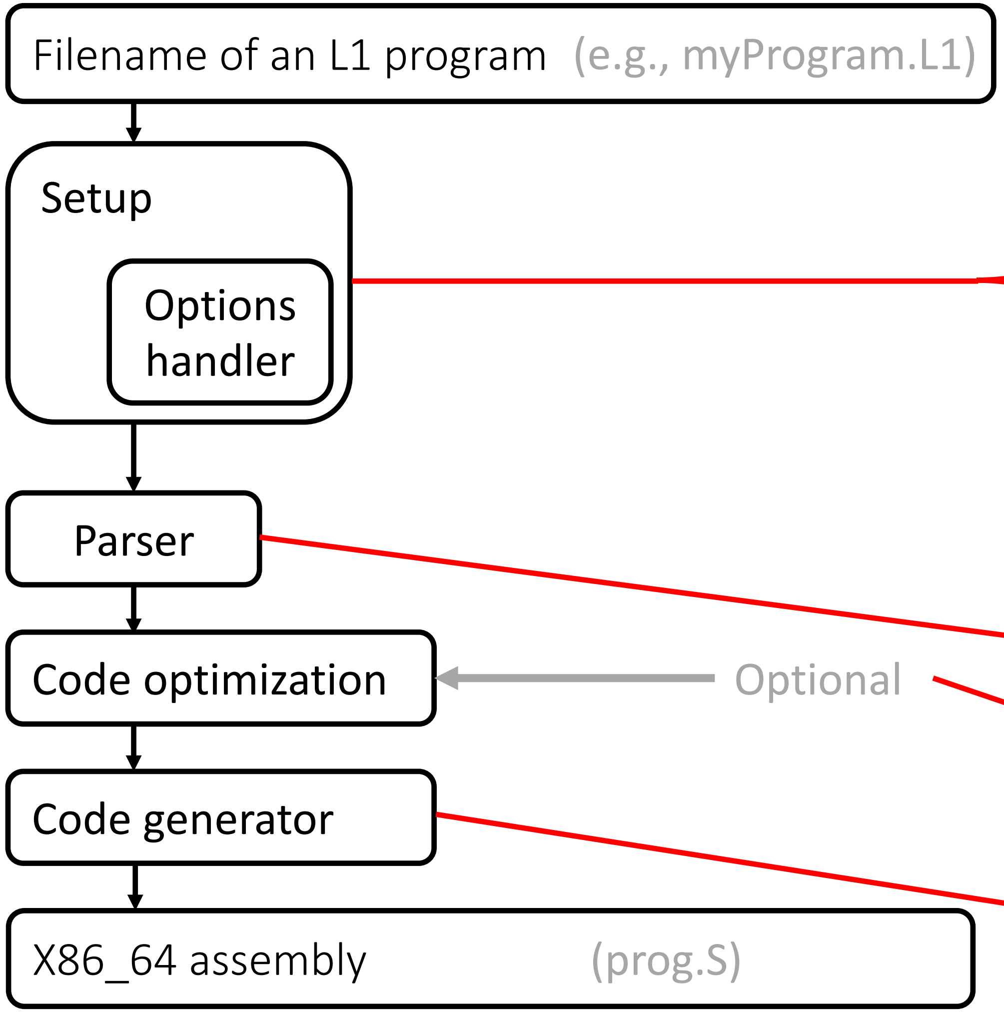
Compiler structure for L1



Show structure in C++ code

- *parsing_examples/0/src/compiler.cpp*

Compiler structure for L1



You can use up to C++17

```
int main(
  int argc,
  char **argv
  ){
  auto enable_code_generator = false;
  int32_t optLevel = 0;
  bool verbose;

  /*
   * Check the compiler arguments.
   */
  if( argc < 2 ) {
    print_help(argv[0]);
    return 1;
  }
  int32_t opt;
  while ((opt = getopt(argc, argv, "vg:O:")) != -1) {
    switch (opt){
      case 'O':
        optLevel = strtoul(optarg, NULL, 0);
        break ;

      case 'g':
        enable_code_generator = (strtoul(optarg, NULL, 0) == 0) ? false : true ;
        break ;

      case 'v':
        verbose = true;
        break ;

      default:
        print_help(argv[0]);
        return 1;
    }
  }

  /*
   * Parse the input file.
   */
  auto p = L1::parse_file(argv[optind]);

  /*
   * Code optimizations (optional)
   */


  /*
   * Generate x86_64 assembly.
   */
  if (enable_code_generator){
    L1::generate_code(p);
  }

  return 0;
}
```

Outline

- Compiler structure
- Parsing
- Parsing with PEG

From L1 to x86_64

Problem:

- Our compiler must recognize
  the structure and the instructions of an L1 program
- However, an L1 program is encoded in a file,
  which can be read as a stream of characters
- How can we recognize an L1 program from a stream of characters?

```
(@go
 (@go
   0 0
   return
  )
)
```

```
(@go\n  (@go\n      0 0\n     return\n   )\n )
```

⇨ L1 compiler

Parsing

It is the process of analyzing a string of symbols (e.g., characters) conforming to the rules of a former grammar.

```
(@go\n  (@go\n      0 0\n     return\n   )\n )
```

- Does this string of symbols represent an L1 program?
- If yes, which L1 program is it?

```
(@go
 (@go
   0 0
   return
  )
)
```

We need a memory representation of the L1 program given as input

Example of memory representation (parsing_examples/7/src/L1.h)

Parsing

It is the process of analyzing a string of symbols conforming to the rules of a former grammar.

```
(@go\n  (@go\n      0 0\n     return\n   )\n )
```

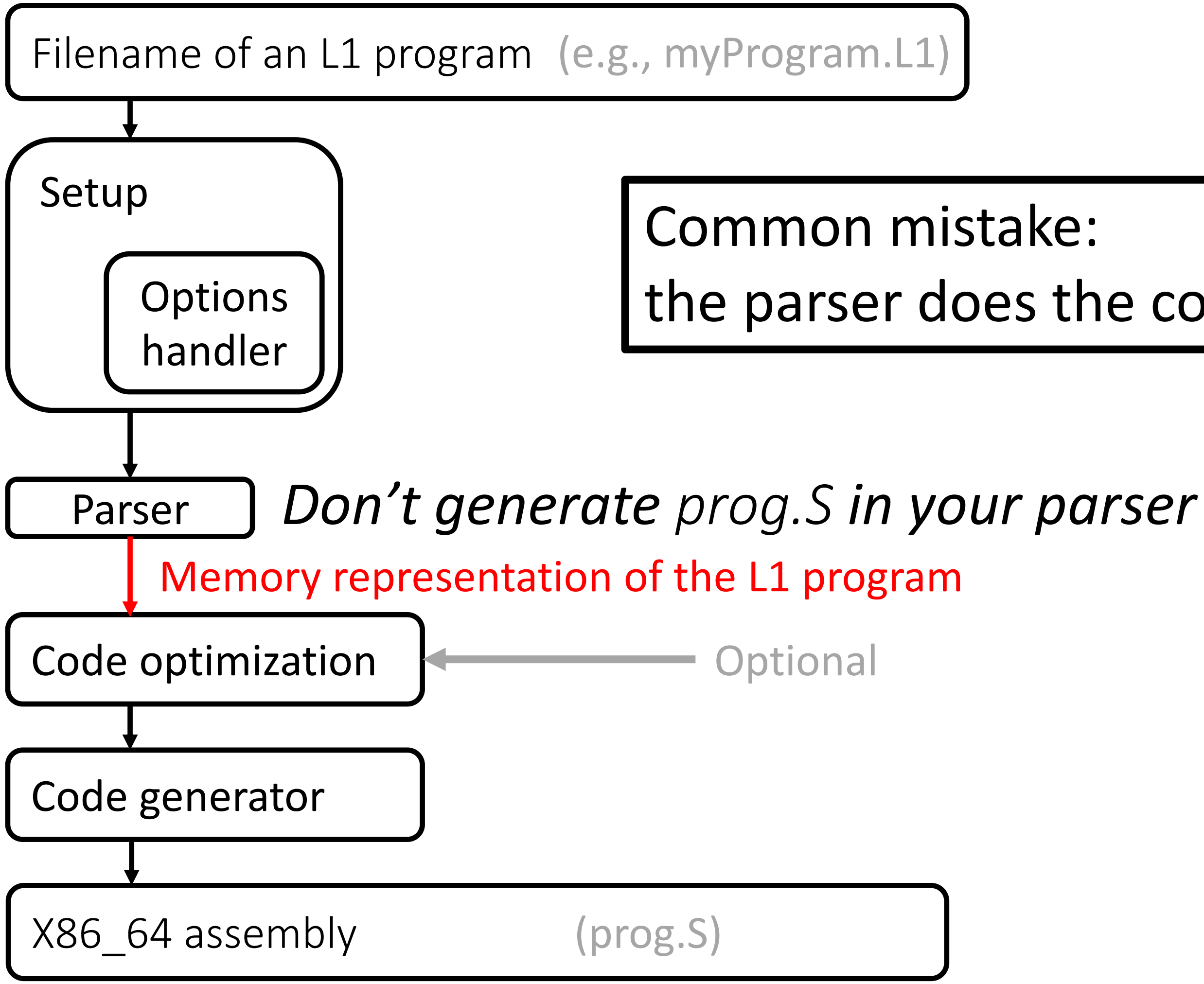
- Does this string of symbols represent an L1 program?
- If yes, which L1 program is it?

We need a memory representation of the L1 program given as input

Example of memory representation (parsing_examples/7/src/L1.h)

```
enum Register {rdi, rax};

class Item {
  public:
    std::string labelName;
    Register r;
    bool isARegister;
};

/*
 * Instruction interface.
 */
class Instruction{
};

/*
 * Instructions.
 */
class Instruction_ret : public Instruction{
};

class Instruction_assignment : public Instruction{
  public:
    Item src,dst;
};

/*
 * Function.
 */
class Function{
  public:
    std::string name;
    int64_t arguments;
    int64_t locals;
    std::vector<Instruction *> instructions;
};

/*
 * Program.
 */
class Program{
  public:
    std::string entryPointLabel;
    std::vector<Function *> functions;
};
```

Compiler structure for L1



Common mistake:
the parser does the code generation

Parser generator

- It generates a parser from its specification
  - Grammar
  - Actions (they are explained next)

- We use Parsing Expression Grammar Template Library (PEGTL) in this class as a parser generator
  - C++ 11
  - Header only
  - Implemented using C++ templates
  - Included in 322_framework/lib/PEGTL
    - 322_framework/lib/PEGTL/lib/PEGTL/src/example/pegtl
    - 322_framework/lib/PEGTL/lib/PEGTL/doc
    - #include <pegtl.hpp>

parsing_examples.tar.bz2

- It contains 8 examples of parsers
  which gradually parse more and more L1 grammar
- The subdirectory "tests" for each example contains
  the files that can be parsed by that example and one that cannot
- This is a good starting point for your L1 parser
- They contain more than a parser
  - They contain code to take compiler inputs (e.g., -O0, -v, -g)
  - They contain an empty code generator that dumps prog.S
  - They contain an almost-empty data structure
    for a memory representation of L1 programs

Designing a parser

- Step 1: define the grammar

Entry point → p ::= (l)

l ::= @name

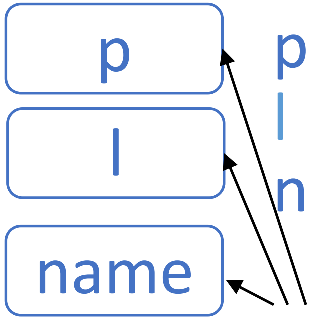
name ::= sequence of chars matching [a-zA-Z_][a-zA-Z_0-9]*

← *Reduction*

(@go)

Designing a parser

- Step 1: define the grammar




```
p    ::= (l)
l    ::= @name
name ::= sequence of chars matching [a-zA-Z_][a-zA-Z_0-9]*
```

- Step 2: define the actions
  - At most one action per grammar rule
  - When a grammar rule is selected, then its action is executed (if the action exists)
  - The actions invoked are responsible to generate the memory representation of the parsed program

Designing a parser

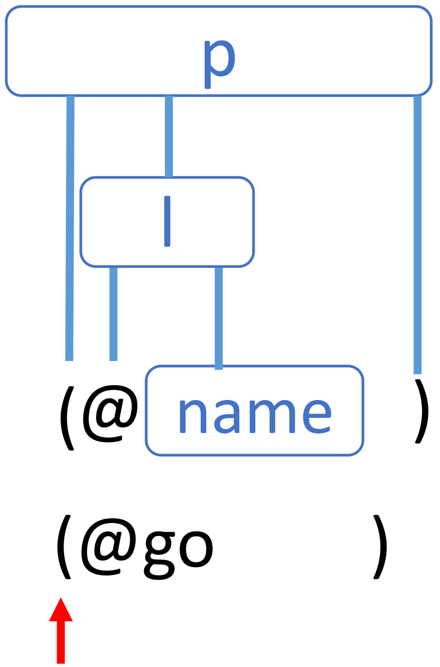
- Step 1: define the grammar

p ::= (l)

l ::= @name

name ::= sequence of chars matching [a-zA-Z_][a-zA-Z_0-9]*

(@go)



Demo time: writing parsers in C++ w/ PEGTL

- *parsing_examples/0/src/parser.cpp*
- *parsing_examples/1/src/parser.cpp*
- *parsing_examples/2/src/parser.cpp*

Actions are invoked bottom up!

Designing a parser (2)

- Step 1: define the grammar

Entry point → p ::= (l f^+)

l ::= @name

f ::= (l)

name ::= sequence of chars matching [a-zA-Z_][a-zA-Z_0-9]*

← *Reduction*

```
(@go
  (@go)
  (@myf1)
  (@myf2)
)
```

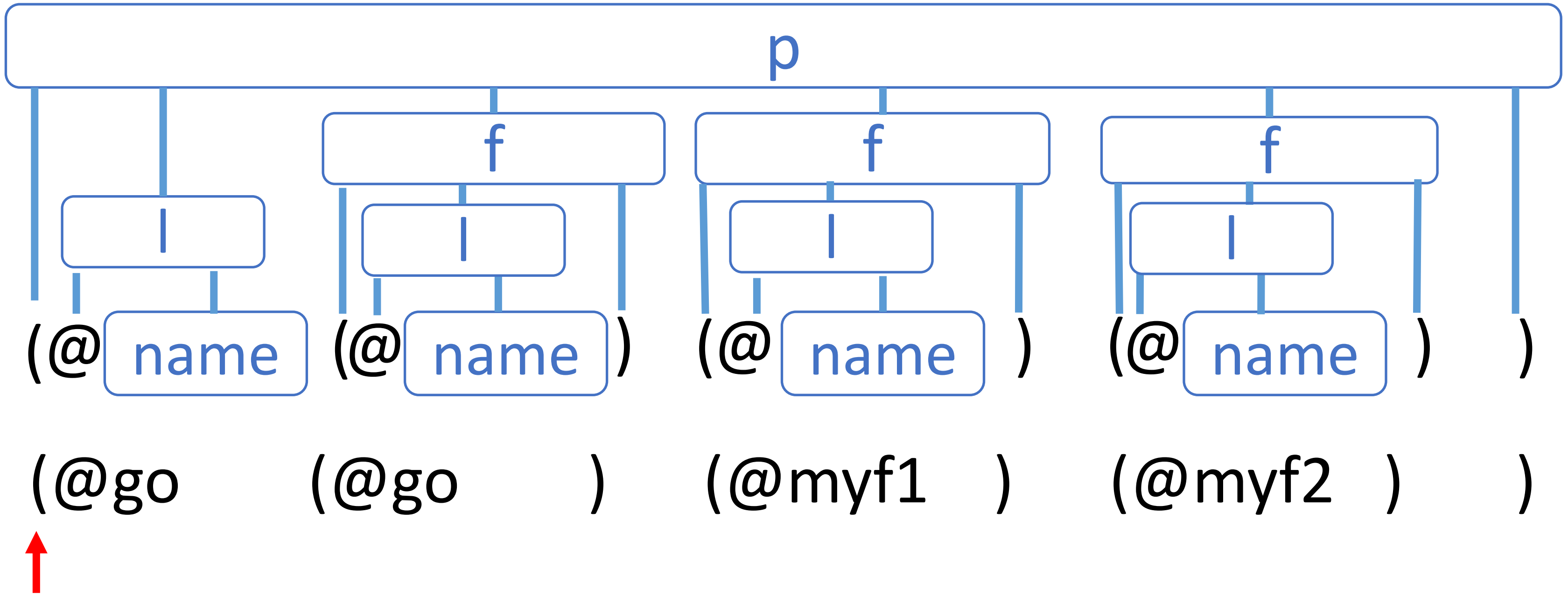
Designing a parser (2)

- Step 1: define the grammar

Entry point →

```
p    ::= (l f+)
l    ::= @name
f    ::= (l)
name ::= sequence of chars matching [a-zA-Z_][a-zA-Z_0-9]*
```

```
(@go
  (@go)
  (@myf1)
  (@myf2)
)
```



Example of an implementation of a parser

- Grammar

  p ::= (l f+)
  f ::= (l)
  l ::= @name
  name ::= [a-zA-Z_][a-zA-Z_0-9]*

- *Actions are invoked bottom up*
- *Hence, at the time we generate l we don't know whether*
  it is
  or

- Stream of tokens
  1. Create a class that represent all possible tokens
  2. Create a stream of tokens (e.g., std::vector<Token *>) s such hat all actions can access it
  3. Actions that generate a token append the just-generated token to s
  4. Actions that generate higher level tokens consume tokens from s and append the higher level one to s

Example of an implementation of a parser

- Grammar

  p ::= (l f⁺)

  f ::= (l)

  l ::= @name

  name ::= [a-zA-Z_][a-zA-Z_0-9]*

- Actions
  - p Create a program p (e.g., instance of the class Program defined in L1.h)
    Add all functions parsed to p by consuming all tokens from s excluding the first one (which is l). Set the entry point of p to be l
  - f Create a new function f (e.g., instance of the class Function defined in L1.h) and set its name to l (taken from the head of s).
    Append f to s (or keep a separate list of functions).
  - l Create a new label l (e.g., instance of the class Label defined in L1.h)
    Add the new label to s. Store the sequence of characters consumed by it
  - No need to set an action for name

Designing a parser

- Does this string of symbols represent an L1 program?
  - If the string of characters is generated by a sequence of grammar rules, then yes

- What is the L1 program encoded in the string of symbols given as input (e.g., `test1.L1`)?
  - Representing the L1 program in memory (`L1.h`) for analysis and/or evaluation is the job of the actions

Outline

- Compiler structure
- Parsing
- Parsing with PEG

Grammar

- Not ambiguous (for programming languages)

- Context Free Grammars

```
INST ::=   VAR <- VAR + VAR
         | VAR <- VAR
```

- Parsing Expression Grammar

```
INST ::=   VAR <- VAR + VAR
         | VAR <- VAR
```

Sequence of actions in PEG

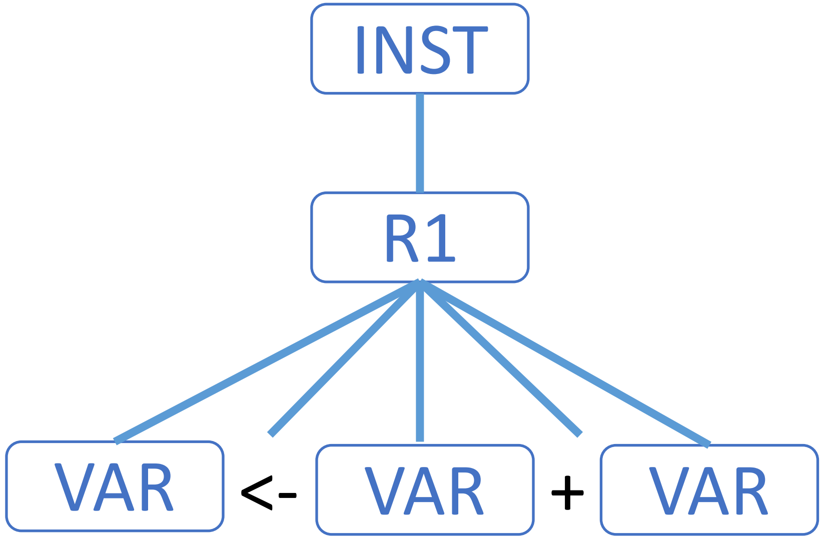
```
INST ::=   VAR <-  VAR + VAR
         | VAR <- VAR
```

Sequence of actions in PEG

R1 ::= VAR <- VAR + VAR
R2 ::= VAR <- VAR

INST ::= R1 | R2

```
struct INST:
   pegtl::sor<
      R1,
      R2
   > { } ;
```



INPUT: “ v5 <- v3 + v1 ”

Actions fired:

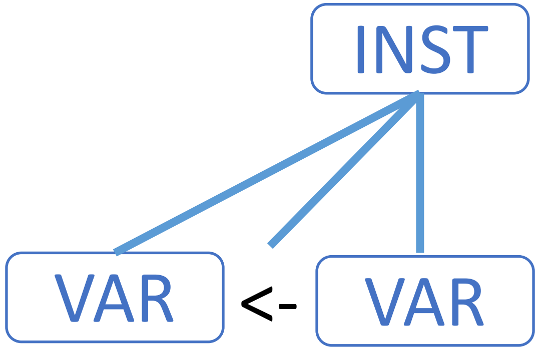
1. VAR
2. <-
3. VAR
4. +
5. VAR
6. R1
7. INST

Sequence of actions in PEG

R1 ::= VAR <- VAR + VAR
R2 ::= VAR <- VAR

INST ::= R1 | R2

```
struct INST:
  pegtl::sor<
    R1,
    R2
  > { } ;
```



INPUT: " v5 <- v3 *"*

Actions fired:

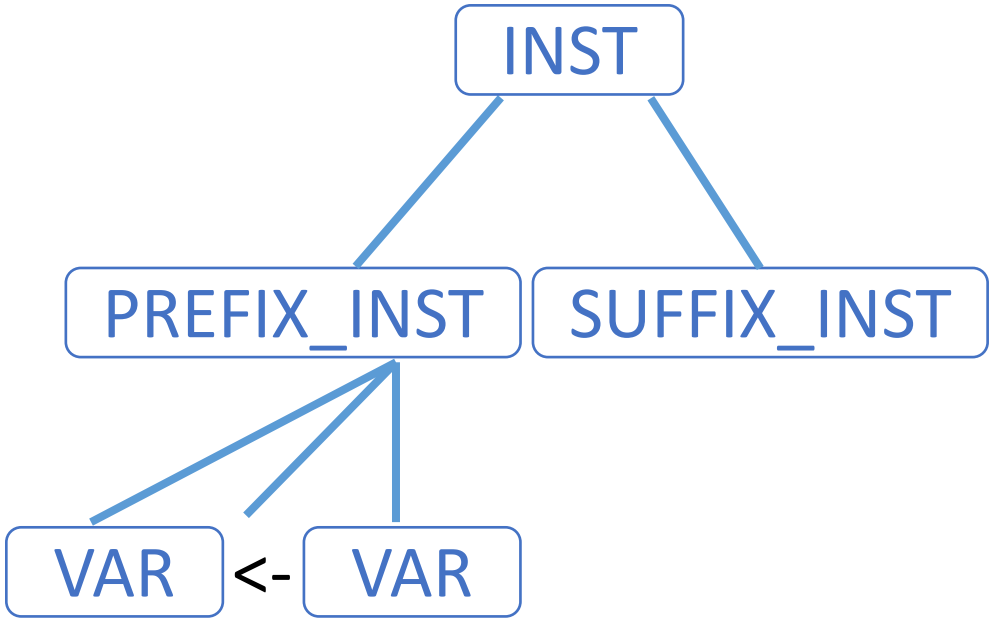
1. VAR
2. <-
3. VAR
4. VAR
5. <-
6. VAR
7. INST

A (too complex) solution for PEG

INST ::= PREFIX_INST SUFFIX_INST

PREFIX_INST ::= VAR <- VAR

SUFFIX_INST ::= “” | + VAR



Actions fired:

1. VAR
2. <-
3. VAR
4. PREFIX_INST
5. SUFFIX_INST
6. INST

INPUT: “ v5 <- v3 ”

A practical solution in PEG

R1 ::= VAR <- VAR + VAR
R2 ::= VAR <- VAR

INST ::= R1 | R2

```
struct INST:
   pegtl::sor<
      R1,
      R2
   > { } ;
```

Actions fired:

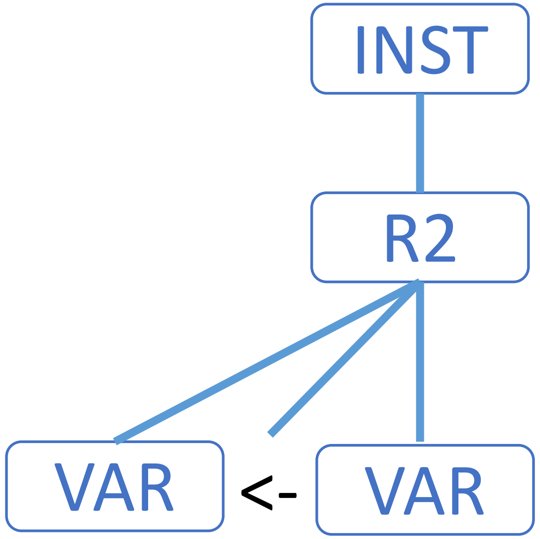
INPUT: “ v5 <- v3 ”

A more practical solution in PEG

R1 ::= VAR <- VAR + VAR
R2 ::= VAR <- VAR

INST ::= R1 | R2

```
struct INST:
   pegtl::sor<
      pegtl::seq<pegtl::at<R1>, R1>,
      pegtl::seq<pegtl::at<R2>, R2>
    > { } ;
```



INPUT: “ v5 <- v3 ”

Actions fired:

1. VAR
2. <-
3. VAR
4. R2
5. INST

Always have faith in your ability

Success will come your way eventually

Best of luck!