Parsing

Simone Campanoni
simonec@eecs.northwestern.edu
Outline

• Compiler structure

• Parsing

• Parsing with PEG
Compiler structure

Program in the source programming language

Setup
  Options handler

Front end

Middle end

Back end

Program in the destination programming language

Optional
Compiler structure for this class

1. Program in the source programming language
2. Setup
   - Options handler
3. Parser
4. Code optimization (Optional)
5. Code generator
6. Program in the destination programming language
Compiler structure for L1

Filename of an L1 program (e.g., myProgram.L1)

Setup
  Options handler

Parser

Code optimization
  Optional

Code generator

X86_64 assembly (prog.S)

Show structure in C++ code
  • parsing_examples/0/src/compiler.cpp
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?
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?**

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

Filename of an L1 program  (e.g., myProgram.L1)

Setup
  Options handler

Parser

Memory representation of the L1 program

Code optimization

Optional

Code generator

X86_64 assembly  (prog.S)

Common mistake: the parser does the code generation on top of parsing and creating the memory representation of a program.
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

\[
p ::= (\text{label})
\]
\[
\text{label} ::= \text{sequence of chars matching } :[a-zA-Z_][a-zA-Z_0-9]^*
\]

(:go)
Designing a parser

• Step 1: define the grammar

p ::= (label)

label ::= 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
  
  \[ p ::= (\text{label}) \]
  
  \[ \text{label} ::= \text{sequence of chars matching } [a-zA-Z_] [a-zA-Z_0-9]^* \]

Actions are invoked bottom up!
Designing a parser

• Step 1: define the grammar
  
  \[ p ::= (\text{label}) \]
  \[ \text{label} ::= \text{sequence of chars matching} : [a-zA-Z_][a-zA-Z_0-9]^* \]

Show a PEGTL parser in C++
- parsing_examples/0/src/parser.cpp
- parsing_examples/1/src/parser.cpp
- parsing_examples/2/src/parser.cpp
Designing a parser (2)

- Step 1: define the grammar

\[ p ::= (\text{label } f^+) \]
\[ f ::= (\text{label}) \]
\[ \text{label} ::= \text{sequence of chars matching } [a-zA-Z_][a-zA-Z_0-9]^* \]

Entry point \( \rightarrow \)

 Reduction

\( (:\text{go} \)
\( (:\text{go}) \)
\( (:\text{myf1}) \)
\( (:\text{myf2}) \)
\)
Designing a parser (2)

• Step 1: define the grammar

\[ p ::= (\text{label } f^+) \]
\[ f ::= (\text{label}) \]
\[ \text{label} ::= \text{sequence of chars matching} :[a-zA-Z_]^[a-zA-Z0-9]* \]
Designing a parser (2)

• Step 1: define the grammar

\[ p ::= (\text{label } f^+) \]
\[ f ::= (\text{label}) \]
\[ \text{label} ::= \text{sequence of chars matching :}[a-zA-Z_][a-zA-Z_0-9]^* \]

Actions are invoked bottom up!
Example of a parser

• Grammar

1. \( p ::= (label ~ f^+) \)
2. \( f ::= (label) \)
3. \( label ::= [:a-zA-Z_][a-zA-Z_0-9]* \)

• Actions

1. Create a program \( p \)
   (e.g., instance of the class Program)
   Add all functions parsed to \( p \)
   Set the entry point of \( p \) to be label

2. Create a new function \( f \) and set its name to \( label \)
   (e.g., instance of the class Function)
   Add \( f \) to the sequence of functions parsed

3. Create a new label \( l \)
   (e.g., instance of the class Label)
   Add the new label to the sequence of labels parsed
   Store the sequence of characters consumed by it

Actions are invoked bottom up!
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
  \[ \text{INST ::= VAR} \leftarrow \text{VAR} + \text{VAR} \]
  \[ \left| \text{VAR} \leftarrow \text{VAR} \right. \]

• Parsing Expression Grammar
  \[ \text{INST ::= VAR} \leftarrow \text{VAR} + \text{VAR} \]
  \[ \left| \text{VAR} \leftarrow \text{VAR} \right. \]
Sequence of actions in PEG

\[
\text{INST ::= } \text{VAR <- VAR + VAR} \\
\quad | \text{VAR <- VAR}
\]
Sequence of actions in PEG

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

INPUT: “v5 <- v3 + v1”

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

struct INST:
  pegtl::sor<
  R1,
  R2
  > { };
Sequence of actions in PEG

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

INPUT: “ v5 <- v3 ”

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

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

INPUT: " v5    <- v3       "

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

```

```
``
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 practical solution in PEG

R1 ::= VAR ← VAR + VAR
R2 ::= VAR ← VAR
INST ::= R1 | R2

INPUT: “ v5 ← v3 ”

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

Actions fired:
1. VAR
2. ←
3. VAR
4. R2
5. INST