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
Compiler structure for this class

- Program in the source programming language
  - Setup
    - Options handler
  - Parser
  - Code optimization (Optional)
  - Code generator
- 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/Simplest/src/compiler.cpp
Outline

• Compiler structure

• Parsing

• Dealing with ambiguity in 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?**

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

Show memory representation in C++ code
(parsing_examples/1/src/L1.h)
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)
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

Show PEGTL simple parsers in C++
• parsing_examples/Simplest/src/parser.cpp
• parsing_examples/Simple/src/parser.cpp
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

\[
p ::= (\text{label})
\]

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

• Step 1: define the grammar
  \[ p ::= (\text{label}) \]
  \[ \text{label} ::= \text{sequence of chars matching } [a-zA-Z_] [a-zA-Z0-9]^* \]

• Step 2: define the actions
  • One action per grammar rule
  • When a grammar rule is selected, then its action is executed
Designing a parser

• Step 1: define the grammar
  
  \[\text{p} ::= (\text{label})\]
  \[\text{label} ::= \text{sequence of chars matching } [a-zA-Z_] [a-zA-Z0-9]*\]
Designing a parser (2)

- Step 1: define the grammar

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

(:go
  (:go)
  (:myf1)
  (:myf2)
)
Designing a parser (2)

• Step 1: define the grammar

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

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

• Step 1: define the grammar

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

Actions will be invoked bottom up!
Example of a parser

• Grammar
  1. \( p ::= (\text{label } f^+) \)
  2. \( f ::= (\text{label}) \)
  3. \( \text{label ::= } [a-zA-Z_.][a-zA-Z_0-9]^* \)

• Actions
  1. Create a program \( p \)
     (e.g., instance of a structure “struct Program”)
     Add all functions parsed to \( p \)
     Set the entry point of \( p \) to be \( \text{label} \)
  2. Create a new function \( f \) and set its name to \( \text{label} \)
     (e.g., instance of a structure “struct Function”)
     Add \( f \) to the sequence of functions parsed
  3. Create a new label \( l \)
     (e.g., instance of a structure “struct 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
- Dealing with ambiguity in PEG
Grammar

• Not ambiguous (for programming languages)

• Context Free Grammars

\[
\text{INST ::= VAR \gets VAR + VAR} \\
| \quad \text{VAR \gets VAR}
\]

• Parsing Expression Grammar

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

\[
\text{INST ::= } \text{VAR} \leftarrow \text{VAR} + \text{VAR} \\
| \text{VAR} \leftarrow \text{VAR}
\]
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”

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

struct INST:
  pegtl::sor<
  R1,
  R2
  > { } ;
A (too complex) solution for PEG

\[
\text{INST} ::= \text{PREFIX\_INST} \text{ SUFFIX\_INST}
\]

\[
\text{PREFIX\_INST} ::= \text{VAR} \leftarrow \text{VAR}
\]

\[
\text{SUFFIX\_INST} ::= "" | + \text{VAR}
\]

**Actions fired:**
1. \text{VAR}
2. \leftarrow
3. \text{VAR}
4. \text{PREFIX\_INST}
5. \text{SUFFIX\_INST}
6. \text{INST}

**INPUT:** “v5 \leftarrow 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 practical solution in PEG

R1 ::= \text{VAR} \leftarrow \text{VAR} + \text{VAR} \\
R2 ::= \text{VAR} \leftarrow \text{VAR} \\
INST ::= R1 \mid R2

INPUT: “v5 \leftarrow v3”

Actions fired:
1. \text{VAR} \\
2. \leftarrow \\
3. \text{VAR} \\
4. \text{R2} \\
5. \text{INST}

\begin{verbatim}
struct INST:
    pegtl::sor<
        pegtl::seq<pegtl::at<R1>, R1>,
        pegtl::seq<pegtl::at<R2>, R2>
    >
\end{verbatim}