Lecture 24: Parsing

Sebastiaan

Video lecture to replace lecture of November 6th

Why the online recording?
An explanation

- Nanny got covid,
  - counting on her being better by Monday
- Daughter tested positive for covid
  - cancelled back-up care to keep back-up care safe
  - my spouse had to teach
  - I had my TAs conduct the midterms
- Nanny will start again on Tuesday
  - spouse has to teach again
  - I’m home with baby
- TLDR: I’m home with the baby, hope this is a rare occurrence

Accessibility note

- If you need (or would like) a transcript for this video, send me an email at sjoosten@umn.edu
- I should get some of the time that goes into preparing videos back by not giving the lecture,
  I can spend it on typing up transcripts if need be!
- Don’t hold back in asking: making a course more accessible ultimately benefits everyone, not just those with DRC letters.

If you have questions

- Use discord as usual
- If you have questions regarding the lecture:
  - Send me an email: sjoosten@umn.edu
- Tag the timestamp in the video you have a question on

Outline

- Project description
- About ASTs, and coming up with them
- How parsers work, and coming up with them
- A bit on lexers (more on Wednesday)
- Some final remarks

Outline

- **Project description**
- About ASTs, and coming up with them
- How parsers work, and coming up with them
- A bit on lexers (more on Wednesday)
- Some final remarks

Project description

- You'll write a program that takes this as input:

```
let (*prove*) cf_idempotent (h : int)
 = (cf (cf h) = cf h) (*hint: axiom *)

let (*prove*) inv_involution (h : int)
 = (inv (inv h) = h) (*hint: axiom *)

let (*prove*) cf_inv_commute (h : int)
 = (cf (inv h) = inv (cf h)) (*hint: axiom *)

let (*prove*) cf_inv_property (h : int)
 = (cf (inv (cf (inv h))) = cf h)
```

Project description

- And prints this:

```
Proof of cf_inv_property:
 cf (inv (cf (inv h)))
= {lemma cf_inv_commute}
 inv (cf (cf (inv h)))
= {lemma cf_idempotent}
 inv (cf (inv h))
= {lemma cf_inv_commute}
 inv (inv (cf h))
= {lemma inv_involution}
 cf h
```

Project description

- We saw:
  - a way to state equalities
  - a way to ask for proofs
  - a proof being produced based on the input
- Your tool will also:
  - accept ocaml definitions
  - treat match statements correctly
  - do induction proofs based on the types

Tool overview

- Input file (supplied by me)
- Parser
- Proof generator (internally calls:)
  - Proof step generator
- Output

Tool overview

- Input file (supplied by me)
- Parser — This week
- Proof generator (internally calls:) — Last week
  - Proof step generator — In-between week
- Output

Lexer, Parser, Driver, Printer, AST

- In your ‘halloween’ lab, you created:
  - a lexer: recognizes words
  - a parser: turns the ‘lexbuf’ into a data-structure
  - a driver: calls the parser and the lexer
- In your upcoming assignment, you’ll:
  - create a data-structure to describe our input, called the abstract syntax tree, or AST
  - create a printer on the AST
  - improve the driver

Outline

- Project description
- **About ASTs, and coming up with them**
- How parsers work, and coming up with them
- A bit on lexers (more on Wednesday)
- Some final remarks

Describing expressions

- My datatype for expressions has three variants:

```
type expression
 = Match of (..)
 | Application of (..)
 | Identifier of string
```

- I’ve made the choice to encode these the same way:
  - ‘Nil’ for a datatype with a Nil constructor
  - ‘x’ for a variable
  - ‘append’ for a function that is defined somewhere
- Not distinguishing these makes parsing and lexing easier
- If need be, these can be distinguished as a second step.

Applications

- Consider these ‘function’ applications:
  - Cons (x, y)
  - foo x
  - bar x y

Applications

- Consider these 'function' applications:
    - Cons (x, y)
    - foo x
    - bar x y
- We can make different choices to capture this:
    - option 1: An application gets a list of arguments:
    - option 2: An application gets a single argument, currying-style

Applications

- Consider these ‘function’ applications:
  - Cons (x, y)
  - foo x
  - bar x y
- We can make different choices to capture this:
  - option 1: An application gets a list of arguments:
  - option 2: An application gets a single argument, currying-style

Applications

- Consider these ‘function’ applications:
  - Cons (x, y)
  - foo x
  - bar x y
- We can make different choices to capture this:
  - option 1: An application gets a list of arguments:
    - Application (cons, [x, y])
    - Application (foo, [x])
    - Application (bar, [x,y])
  - option 2: An application gets a single argument, currying-style
    - Application (cons, Tuple [x, y])
    - Application (foo, x)
    - Application (Application (bar, x), y)

- let cons = Identifier “Cons”
- … and so on

Applications

- option 1: An application gets a list of arguments:
  - Application (cons, [x, y])
  - Application (foo, [x])
  - Application (bar, [x,y])
- Q: What is the type of the Application constructor in this case?

Applications

- option 1: An application gets a list of arguments:
  - Application (cons, [x, y])
  - Application (foo, [x])
  - Application (bar, [x,y])
- Q: What is the type of the Application constructor in this case?
- A1: Application of (expression, expression list)
- A2: Application of (string, expression list)

Applications

- option 2: An application gets a single argument, currying-style
  - Application (cons, Tuple [x, y])
  - Application (foo, x)
  - Application (Application (bar, x), y)
- Q: What is the type of the Application constructor in this case?

Applications

- option 2: An application gets a single argument, currying-style
    - Application (cons, Tuple [x, y])
    - Application (foo, x)
    - Application (Application (bar, x), y)
- Q: What is the type of the Application constructor in this case?
- A1: Application of (expression * expression)
- A2: What is the type of ‘Tuple’ ?

Applications

  - option 2: An application gets a single argument, currying-style
    - Application (cons, Tuple [x, y])
    - Application (foo, x)
    - Application (Application (bar, x), y)
- This would have you use the following type for expressions:
- type expression = | …
  | Application of (expression * expression)
  | Tuple of (expression list)

How to choose a good AST?

- Choose something that makes sense *to you*.
- Don’t be afraid of needing to refactor your AST:
  - It’s not so difficult in ocaml
  - It’s not as error-prone in ocaml
  - … provided you use plenty of constructors!

- I cannot stress the importance of constructors too much…
  - they increase readability
  - they help with type error messages
  - they give your editor something to use ‘find’ on

How to come up with an AST?

1. Take a look at some syntax:
   - foo x y
   - Cons (h,tl)
2. Write the values as ***you*** expect the ocaml values to be:
   - Application (Application (foo,[x]),[y])
   - Application (cons, [h,tl])
3. Determine the type of the values you wrote

(While you’re at it, why not write a to-string function?)

Outline

- Project description
- About ASTs, and coming up with them
- **How parsers work, and coming up with them**
- A bit on lexers (more on Wednesday)
- Some final remarks

A parser

- Here is the parser you did on halloween (improved syntax):

```
%token <string> WORD
%token EOF
%start main
%type <string list> main
%%
main:
| l = words ; EOF { l }
words:
| { [] }
| w = WORD; l = words { w :: l }
```

main: indicates that this is a parsing rule
This almost always has a single variant,
ending with EOF (end of file)

words: also a parsing rule.
The first variant accepts nothing

A parser

- A list of tokens is passed to the parser ‘main’
  WORD(“hello”); WORD(“world”); EOF
- The parser then tries to fit it to one of the ‘main’ variants:
  - ‘words’ can match: WORD(“hello”); WORD(“world”)

```
%token <string> WORD
%token EOF
main:
| l = words ; EOF { l }
words:
| { [] }
| w = WORD; l = words { w :: l }
```

A parser

- The empty string (no tokens) is a ‘words’ and its value is []
- we can add a WORD token in front of a ‘words’,
  if the WORD has value w, and ‘words’ value ‘l’, then the combined value is w::l
  - so WORD(“world”) has the value “world” :: []
  - and WORD(“hello”);WORD(“world”)
    has the value “hello”::(“world”::[])

```
words:
| { [] }
| w = WORD; l = words { w :: l }
```

A parser

- We could've also used the list 'function' (on rules)
- This is the same parser:

```
%token <string> WORD
%token EOF
main:
| l = list(WORD) ; EOF { l }
```

Coming up with a parser

- This follows roughly the same process, but we need to consider each possible syntax:

```
expression:
| lhs = expression ;
  arg = IDENT
  { Application (… depends on data-type …) }
| lhs = expression ; LPAREN ;
  args = separated_nonempty_list(COMMA, expression) ;
  RPAREN
   { … depends on data-type … }
| nm = IDENT { Identifier nm }
```

Parsing

- This follows roughly the same process, but we need to consider each possible syntax:

we describe how to parse an expression, there are three variants

```
expression:
| lhs = expression ;
  arg = IDENT
  { Application (… depends on data-type …) }
| lhs = expression ; LPAREN ;
  args = separated_nonempty_list(COMMA, expression) ;
  RPAREN
   { … depends on data-type … }
| nm = IDENT { Identifier nm }
```

Parsing

- This follows roughly the same process, but we need to consider each possible syntax:

In the first variant, the lhs is an expression, like ‘bar x’, the arg is a simple identifier, like ‘y’. This variant says we can put them together

```
expression:
| lhs = expression ;
  arg = IDENT
  { Application (… depends on data-type …) }
| lhs = expression ; LPAREN ;
  args = separated_nonempty_list(COMMA, expression) ;
  RPAREN
   { … depends on data-type … }
| nm = IDENT { Identifier nm }
```

Parsing

- This follows roughly the same process, but we need to consider each possible syntax:

```
expression:
| lhs = expression ;
  arg = IDENT
  { Application (… depends on data-type …) }
| lhs = expression ; LPAREN ;
  args = separated_nonempty_list(COMMA, expression) ;
  RPAREN
   { … depends on data-type … }
| nm = IDENT { Identifier nm }
```

In the second variant, the lhs is an expression,
like ‘bar x’, but then there are parentheses.
This allows us to pass whole expressions as arguments.

Parsing

- This follows roughly the same process, but we need to consider each possible syntax:

```
expression:
| lhs = expression ;
  arg = IDENT
  { Application (… depends on data-type …) }
| lhs = expression ; LPAREN ;
  args = separated_nonempty_list(COMMA, expression) ;
  RPAREN
   { … depends on data-type … }
| nm = IDENT { Identifier nm }
```

The last variant is a single identifier, like ‘bar’, or ‘x’

Parser syntax

- Parser variants are separated by |
- Within each variant, I can write a sequence of what needs to occur that. I can put a parser or a token there.
  - Parsers are written in lower case letters (like ‘expression’)
  - Tokens are upper case (like IDENT)
  - I like to use ; to separate the tokens, but that’s *somewhat* optional
- Some tokens contain a value (like IDENTIFIER), and all parsers do, too. We can use these to construct our AST.

Constructing the AST

- Here’s how to construct the AST for ‘option 2’

```
| lhs = expression ;
  arg = IDENT
  { Application (lhs, Identifier arg) }
```

Note how we use the values ‘lhs’ and ‘arg’ in our grammar

Outline

- Project description
- About ASTs, and coming up with them
- How parsers work, and coming up with them
- **A bit on lexers** (more on Wednesday)
- Some final remarks

Getting tokens (on Lexers)

- Once you've written a part of your parser, how do you get tokens in it?
- My lexer is as the halloween example, but includes something like this:

```
rule token =
 | [' ' '\t'] { token lexbuf }
 | "(*prove*)" { PROVE }
 | "(*hint:" { HINT }
 | "(*" { comment 0 lexbuf }
 …
 |
 | ['a'-'z' 'A'-'Z' '0'-'9' '?' '_' '\'']+ as id
    { IDENT id }
```

Getting tokens

- Once you've written a part of your parser, how do you get tokens in it?
- My lexer is as the halloween example, but includes something like this:

The first thing that matches is used, so (*prove*) prevents (* from opening a comment

```
rule token =
 | [' ' '\t'] { token lexbuf }
 | "(*prove*)" { PROVE }
 | "(*hint:" { HINT }
 | "(*" { comment 0 lexbuf }
 …
 |
 | ['a'-'z' 'A'-'Z' '0'-'9' '?' '_' '\'']+ as id
    { IDENT id }
```

I'll say more about 'comment 0 lexbuf' later

This catches any nonempty string with the characters a-z, A-Z, 0-9, ?, _, or ', and calls it an IDENT

Outline

- Project description
- About ASTs, and coming up with them
- How parsers work, and coming up with them
- A bit on lexers (more on Wednesday)
- **Some final remarks**

Final remarks

- Ambiguous grammars
- Parsing from a file
- Starting small

Ambiguous grammars

- Sometimes a parser can have an ambiguous grammar:
  - expression =
    | e1 = expression ; e2 = expression
      {Application (e1,e2)}
    | nm = IDENT {Identifier nm}
- Now ‘foo x y’ can be parsed as:
  Application (foo, Application (x, y))
  Application (Application (foo,x), y)
- menhir will warn you about this, **don’t ignore the warnings**!
  (you only get them when running ‘dune build’ after changes to your parser)
- The grammar for your project is designed not to be ambiguous
  - Nested match statements shouldn’t be allowed, or at least not without extra parentheses!

Parsing from a file…

- I’ll add some code for this later (before lab)
- Check for updates on the partial-submission description

.. starting small

- start with a file that just has:
  cf (cf h)
- … and parse it as an expression, try printing it back
- next, try: (cf (cf h) = cf h)
- … and parse it as an equality (or whatever your datatype is for this)
- next, try:
  `let (*prove*) cf_idempotent (h : int) = (cf (cf h) = cf h)`
- … and parse it as a list of declarations
- … then try the gettingstarted.ml file without comments
- … then try the gettingstarted.ml file as supplied

Hope to see you soon …

- I hope to see you in person again on Wednesday!
- … I’m trying to set up this video s.t. watching it gets you signed off as if this was a quiz, but I’ll not spend too much time on it.