Rackette!
Big picture

# rackette "(define a 3) (+ a 1)" ;;
#:string: "4"
#

• rackette consumes a string that contains a Rackette program, and produces the printed-representation of the result of evaluating the final (if any) expression.
Rackette language

• Data types:
  • int
  • bool
  • proc

• No strings

• Other "features"
  • No comments
  • program is a sequence of definitions followed by zero or one expressions
  • relatively few builtins
Syntax

- define
- cond
- if
- true, false
- and, or
- lambda
- let
- empty
- Optional: let*, letrec
Builtins

• Arithmetic: +, -, *, / (i.e., quotient), remainder
• Boolean: not
• Lists: cons, cons?, empty?, first, rest
• Optional (?): list
REPL

- Read-Eval-Print loop
- Expanded: Read-and-parse/process/print loop
"read" phase (really "read and parse")

• Generally reading programs and checking that they match a given syntax is messy; uses specialized tools.
• For Racket, we can do it in two steps
  • Initially check that input matches BNF for a simpler language
  • Then refine
Rackette sequence

- read: raw text to "concrete program" (Num, List, Symbol)
- parse: concrete program to abstract program (if-expression, definition, cond-expression, ...)
- process: handle definitions, evaluate expressions
  - Evaluate: expressions to values (VInt, VBuiltin, VClosure)
- print: print out values
Concrete programs (what "read" produces)

type concrete_program_piece =
    | Number of int
    | Symbol of string
    | List of (concrete_program_piece list)
type concrete_program = concrete_program_piece list
Concrete programs

type concrete_program_piece =
  | Number of num
  | Symbol of string
  | List of (concrete_program_piece list)
type concrete_program = concrete_program_piece list

• read "(+ 1 2)";;
  • [List [Symbol "+"; Number 1; Number 2]]

• read "(define a 5)";;
  • Failure: syntax error

• read "(define a 5)";;
  • [List [Symbol "define"; Symbol "a"; Number 5]]

• read "(define a 5) a";;
  • [List [Symbol "define"; Symbol "a"; Number 5]; Symbol "a"]

• read "(3 if define)";;  ⇒ Nonsense as Racket; perfectly OK ConcreteProgram!
  • [List [Number 3; Symbol "if"; Symbol "define"]]

รถไฟฟ้า
• Checking to see if a concrete program corresponds to Rackette syntax, and identifying the kind of syntax each piece of the program is (e.g., a name-definition, a function-definition, an if-expression, ...)

• As usual, we break this into smaller tasks

• Parse *each piece* of a concrete program

• Code:

```haskell
parse (input: concrete_program) : abstract_program =
    List.map parse_piece input;;
```

• Of course, we now have to write parse_piece!
Parsing: parse_piece

• Input: a concrete program piece
• Output: a representation of what part of a rackette program we've got (if any!)
• Need a data type for the output!

type concrete_program_piece =
  | Number of num
  | Symbol of string
  | List of (concrete_program_piece list)
From lecture 6 (slightly modified)

\[
\begin{align*}
<\text{prog}> & := <\text{defn}>* \ [ \ <\text{top-level-expr}> \ ] \\
<\text{defn}> & := <\text{name-defn}> \ | \ <\text{proc-defn}> \\
<\text{top-level-expr}> & := <\text{expr}> \\
<\text{name-defn}> & := (\ \text{define} \ <\text{name}> \ <\text{expr}> \ ) \\
<\text{proc-defn}> & := (\ \text{define} \ (\ <\text{name}> \ <\text{name}>* \ ) \ <\text{expr}> \ ) \\
<\text{expr}> & := <\text{number-expr}> \ | \ <\text{string-expr}> \ | \ <\text{name-expr}> \ | \ <\text{bool-expr}> \ | \ <\text{proc-app-expr}> \\
& \quad \ | \ <\text{and-expr}> \ | \ <\text{or-expr}> \ | \ <\text{if-expr}> \ | \ <\text{lambda-expr}> \ | \ <\text{let-expr}> \ | \\
<\text{cond-expr}> & := (\ \text{cond} \ <\text{else-pair}> \ ) \ | \ (\ \text{cond} \ <\text{cond-pair}> \ <\text{cond-pair}>* \ [ \ <\text{else-pair} \ ] \ ) \\
<\text{else-pair}> & := [ \ \text{else} \ <\text{expr}> \ ] \\
<\text{cond-pair}> & := [ \ <\text{expr}> \ <\text{expr}> \ ] \\
\end{align*}
\]

\textit{lambda, let, omitted for space}
Parsing details: what's an abstract program?

- An ocaml representation of a Rackette program
  - by "rackette program", we mean "a concrete program that matches the Rackette Syntax"

```ocaml
type ident = Ident of string

type expression = ...

type definition = ident * expression
  (* only handles name-defs! *)

type abstract_program_piece =
  | Def of definition
  | TLExp of expression

type abstract_program = abstract_program_piece list
```
Deeper details: what's an expression?

type expression =
| IfE of if_expr
| AndE of and Expr
| ...
| ...
| ...
if_expr = expression * expression * expression
and
and_expr = ....
What does parse_piece look like?

Matching!

```ocaml
parse_piece (input: concrete_program_piece) :
abstract_program_piece =

match input with:
| Number n -> ...
| Symbol s -> ...
| List lst -> <this is the interesting part!>

type ident = Ident of string
type expression = ...
type definition = ident * expression
type abstract_program_piece =
    | Def of definition
    | TLExp of expression
type abstract_program = abstract_program_piece list
```
What does parse_piece look like? Matching!

```ocaml
define parse_piece (input: concrete_program_piece) : abstract_program_piece =
    match input with:
    | Number n -> TLExp ( NumE (n) )
    | Symbol s -> TLExp ( ... )
    | List lst -> <this is the interesting part!>
```

type ident = Ident of string
type expression = ...
type definition = ident * expression
type abstract_program_piece =
    | Def of definition
    | TLExp of expression
type abstract_program = abstract_program_piece list
Details

| List lst -> <this is the interesting part!>

| List lst -> match lst with
  | [] -> failwith "Empty parenthesized expression not allowed."
  | (Symbol "define") :: tl -> match tl with ...
  | (Symbol "if") :: exp1 :: exp2 :: exp3 -> ...
  | (Symbol "if") :: _ -> failwith "Malformed if-expression."
  | ...

Digging deeper

| List lst -> match lst with
| ...
|  (Symbol "if") :: exp1 :: exp2 :: exp3 ->
|    IfE [(parse_piece exp1); (parse_piece exp2); (parse_piece exp3)]
|  ...

Why is parsing tough?

• Tons of cases
  • if-expressions
  • and-expressions
  • cond-expressions

• Error-reporting is challenging (but essential!)

• Keeping straight the difference between concrete_program pieces and abstract_program pieces.
  • OCaml's type-checking is helpful here
Why is parsing easy?

• No need to worry about meaning
  • Is the first expression in an if-expression a boolean? We don't care! (yet...)
Suppose you've got an abstract program

```haskell
type abstract_program_piece =
    | Def of definition
    | TLExp of expression
```

How do you process it?

One piece at a time!
Processing (2)

• Remember parsing?

```haskell
parse (input: concrete_program) :
abstract_program =
    List.map parse_piece input;;
```

• Not quite as simple as that

• We need both a program and an environment

• Each step of processing either changes the environment ("define"!) or produces a value (evaluate a top-level expression)

• Handling definitions involves evaluating expressions...so let's discuss evaluation
Evaluation

- We evaluate expressions in the presence of an environment in which we can look things up.
- Usually that's the top level environment, but sometimes it's the TLE plus some "local environment" that has additional bindings.
- We'll just pass in two environments: the TLE and the local.

\[
\text{eval: environment } \times \text{ environment } \times \text{ expr } \rightarrow \text{ value}
\]
Basic evaluation: a huge match-case!

let rec eval envts-omitted (input : abstract_program) =
match input with
| NumE (n) ->
| ProcAppE (...) ->
Basic evaluation: a huge match-case!

let rec eval envts-omitted (input : abstract_program) =
match input with
| NumE (n) -> VNum n
| ProcAppE (...) ->
Printing: the easy part!

let rec print (myVal : value) =
match input with
| VNum n -> string_of_int n
| VBool true -> "true"
| VBool false -> "false"
| VBuiltin ... ->
| ...
Alternative version

• An alternative version of "print" causes the output to appear on the terminal, but doesn't generate an actual *string*

• Not what we'll do in Rackette, but we use these ideas in Game, so we'll present them here, pretending that this is what we want.
Printing: the easy part!

- Wait a minute...."print" doesn't produce a value, it prints something out!
- What "type" does "print something out" have?
  \[\text{print\_int}: \text{int} \rightarrow \text{unit}\]
- unit is a new type for us: it's the type for "no value at all"
- \text{print\_int} is special too: it has a \textit{side effect}, which is displaying something in the window in which you're running OCaml
Printing: the easy part!

let rec print (myVal : value) =
match input with
| VNum n -> ...
| VBuiltin ... ->
| ...
Printing: the easy part!

let rec print (myVal : value) : unit =
match input with
| VNum n -> ...
| VBuiltin ... -> ...
| ...
Printing: the easy part!

let rec print (myVal : value) : unit =
match input with
| VNum n -> print_int n; print_string "\n"
| VBuiltin ... ->
| ...
Wait a minute! What's that semicolon doing?

- "Semicolon" is an operator in Ocaml, just like + or *
- It's an infix operator, of type 'a * 'b -> 'b
- It works by evaluating p and q, and then producing the value of q
- It generates a warning if p is not of type unit
- Most frequent use: printing stuff out like this!