About compilation

- Compilation
  - Analyzer
    - Scanning (lexical scanning)
      - Generates – Identifying different things (colored differently)
        - Lexemes
        - Lexical items
        - Tokens

        means something
    - Parsing – structure according to language's syntax rules. understands program's structure and meaning
      - Generates
        - AST - Abstract Syntax Tree
        - Syntactic structure
        - Grammatical structure
  - Translator abstract syntax tree → machine code
- All this work simplified → general purpose tool – not spesific language → takes grammar of language ↓ can analyze and translate depending on that given language
  - Lexical analyzers (lex)
  - Parser generators (yacc) yet another compiler
  - Use scheme ☺

C language

type definition, for grouping

```
int main()
{
        printf("hello, world");
        return 0;
}
```

built-in function, string, keyword, source code

Plumming: That's something has to be done but I do not wanna do it.

Defining Interpreter

- Define formally semantic for each expression (behavior spesification)
- ? Implementation

An example program

- Input

```
"-(55, -(x,11))"
```

text file

send to parser $\xrightarrow{\text{get}}$ abstract syntax tree

- Scanning & parsing

· defined data type structure

```
(scan&parse "-(55, -(x,11))")
```

- The AST · Abstract Syntax Tree

```
#(struct:a-program
  #(struct:diff-exp
    #(struct:const-exp 55)
    #(struct:diff-exp
      #(struct:var-exp x)
      #(struct:const-exp 11))))
```

→ This is a program

consists of

#(struct:const-exp 55) → constant expression

#(struct:diff-exp → difference expression

#(struct:var-exp x) → variable expression

#(struct:const-exp 11)))) → constant expression

Program ::= *Expression*

$\boxed{\texttt{a-program (exp1)}}$

Expression ::= *Number*

$\boxed{\texttt{const-exp (num)}}$

Expression ::= `-` (*Expression* , *Expression*)

$\boxed{\texttt{diff-exp (exp1 exp2)}}$

Expression ::= `zero?` (*Expression*)

$\boxed{\texttt{zero?-exp (exp1)}}$

Expression ::= `if` *Expression* `then` *Expression* `else` *Expression*

$\boxed{\texttt{if-exp (exp1 exp2 exp3)}}$

Expression ::= *Identifier*

$\boxed{\texttt{var-exp (var)}}$

Expression ::= `let` *Identifier* `=` *Expression* `in` *Expression*

$\boxed{\texttt{let-exp (var exp1 body)}}$

Lecture 12
Let – Implementation

T. METIN SEZGIN

Specifying the behavior

- Programs

```
(value-of-program exp)
= (value-of exp [i=⌈1⌉,v=⌈5⌉,x=⌈10⌉])
```

- Expressions
  - Constructors

const-exp	$: Int \to Exp$
zero?-exp	$: Exp \to Exp$
if-exp	$: Exp \times Exp \times Exp \to Exp$
diff-exp	$: Exp \times Exp \to Exp$
var-exp	$: Var \to Exp$
let-exp	$: Var \times Exp \times Exp \to Exp$

```
(value-of (const-exp n) ρ) = (num-val n)
(value-of (var-exp var) ρ) = (apply-env ρ var)
```

```
(value-of (diff-exp exp1 exp2) ρ)
= (num-val
    (-
      (expval->num (value-of exp1 ρ))
      (expval->num (value-of exp2 ρ))))
```

  - Observer

value-of	$: Exp \times Env \to ExpVal$

Specifying the behavior

- Programs

$$\texttt{(value-of-program } exp\texttt{)} = \texttt{(value-of } exp\ \texttt{[i=}\lceil 1\rceil\texttt{,v=}\lceil 5\rceil\texttt{,x=}\lceil 10\rceil\texttt{])}$$

- Expressions
  - Constructors

const-exp	$: Int \to Exp$
zero?-exp	$: Exp \to Exp$
if-exp	$: Exp \times Exp \times Exp \to Exp$
diff-exp	$: Exp \times Exp \to Exp$
var-exp	$: Var \to Exp$
let-exp	$: Var \times Exp \times Exp \to Exp$

$$\frac{\texttt{(value-of } exp_1\ \rho\texttt{)} = val_1}{\begin{array}{l}\texttt{(value-of (zero?-exp } exp_1\texttt{) } \rho\texttt{)} \\ = \begin{cases}\texttt{(bool-val \#t)} & \text{if } \texttt{(expval->num } val_1\texttt{)} = 0 \\ \texttt{(bool-val \#f)} & \text{if } \texttt{(expval->num } val_1\texttt{)} \neq 0\end{cases}\end{array}}$$

$$\frac{\texttt{(value-of } exp_1\ \rho\texttt{)} = val_1}{\begin{array}{l}\texttt{(value-of (if-exp } exp_1\ exp_2\ exp_3\texttt{) } \rho\texttt{)} \\ = \begin{cases}\texttt{(value-of } exp_2\ \rho\texttt{)} & \text{if } \texttt{(expval->bool } val_1\texttt{)} = \texttt{\#t} \\ \texttt{(value-of } exp_3\ \rho\texttt{)} & \text{if } \texttt{(expval->bool } val_1\texttt{)} = \texttt{\#f}\end{cases}\end{array}}$$

  - Observer

value-of	$: Exp \times Env \to ExpVal$

Specifying the behavior

- Programs

$$\texttt{(value-of-program } exp\texttt{)} = \texttt{(value-of } exp \; \texttt{[i=}\lceil 1 \rceil\texttt{,v=}\lceil 5 \rceil\texttt{,x=}\lceil 10 \rceil\texttt{])}$$

- Expressions
  - Constructors

const-exp	$: Int \to Exp$
zero?-exp	$: Exp \to Exp$
if-exp	$: Exp \times Exp \times Exp \to Exp$
diff-exp	$: Exp \times Exp \to Exp$
var-exp	$: Var \to Exp$
let-exp	$: Var \times Exp \times Exp \to Exp$

$$\frac{\texttt{(value-of } exp_1 \; \rho) = val_1}{\begin{array}{l}\texttt{(value-of (let-exp } var \; exp_1 \; body\texttt{)} \; \rho\texttt{)} \\ \quad = \texttt{(value-of } body \; [var = val_1]\rho\texttt{)}\end{array}}$$

$$\begin{array}{l}\texttt{(value-of (let-exp } var \; exp_1 \; body\texttt{)} \; \rho\texttt{)} \\ = \texttt{(value-of } body \; \texttt{[}var\texttt{=(value-of } exp_1 \; \rho\texttt{)]} \, \rho\texttt{)}\end{array}$$

  - Observer

value-of	$: Exp \times Env \to ExpVal$

Behavior implementation

what we envision

Let $\rho =$ `[i=1,v=5,x=10]`.

```
(value-of
  <<-(-(x,3), -(v,i))>>
  ρ)

= ⌈(-
    ⌊(value-of <<-(x,3)>> ρ)⌋
    ⌊(value-of <<-(v,i)>> ρ)⌋)⌉

= ⌈(-
    (-
      ⌊(value-of <<x>> ρ)⌋
      ⌊(value-of <<3>> ρ)⌋)
    ⌊(value-of <<-(v,i)>> ρ)⌋)⌉

= ⌈(-
    (-
      10
      ⌊(value-of <<3>> ρ)⌋)
    (value-of <<-(v,i)>> ρ))⌉
```

```
= ⌈(-
    (-
      10
      3)
    ⌊(value-of <<-(v,i)>> ρ)⌋)⌉

= ⌈(-
    7
    ⌊(value-of <<-(v,i)>> ρ)⌋)⌉

= ⌈(-
    7
    (-
      ⌊(value-of <<v>> ρ)⌋
      ⌊(value-of <<i>> ρ)⌋))⌉

= ⌈(-
    7
    (-
      5
      ⌊(value-of <<i>> ρ)⌋))⌉
```

```
= ⌈(-
    7
    (-
      5
      1))⌉

= ⌈(-
    7
    4)⌉

= ⌈3⌉
```

Behavior implementation

what we envision

Let ρ = [x=⌈33⌉,y=⌈22⌉].

```
(value-of
  <<if zero?(-(x,11)) then -(y,2) else -(y,4)>>
  ρ)

= (if (expval->bool (value-of <<zero?(-(x,11))>> ρ))
    (value-of <<-(y,2)>> ρ)
    (value-of <<-(y,4)>> ρ))

= (if (expval->bool (bool-val #f))
    (value-of <<-(y,2)>> ρ)
    (value-of <<-(y,4)>> ρ))

= (if #f
    (value-of <<-(y,2)>> ρ)
    (value-of <<-(y,4)>> ρ))

= (value-of <<-(y,4)>> ρ)

= ⌈18⌉
```

Nugget

Intro to implementation

It all revolves around **`value-of`**

The Interpreter

run : *String* → *ExpVal*

```
(define run
  (lambda (string)
    (value-of-program (scan&parse string))))
```

value-of-program : *Program* → *ExpVal*

```
(define value-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1)
        (value-of exp1 (init-env))))))
```

The Interpreter

value-of : $Exp \times Env \rightarrow ExpVal$

```
(define value-of
  (lambda (exp env)
    (cases expression exp
```

$$\boxed{\texttt{(value-of (const-exp } n\texttt{) } \rho\texttt{) = } n}$$

```
      (const-exp (num) (num-val num))
```

$$\boxed{\texttt{(value-of (var-exp } var\texttt{) } \rho\texttt{) = (apply-env } \rho \ var\texttt{)}}$$

```
      (var-exp (var) (apply-env env var))
```

$$\boxed{\begin{array}{l}\texttt{(value-of (diff-exp } exp_1 \ exp_2\texttt{) } \rho\texttt{) =} \\ \quad \lceil \texttt{(- } \lfloor \texttt{(value-of } exp_1 \ \rho\texttt{)} \rfloor \ \lfloor \texttt{(value-of } exp_2 \ \rho\texttt{)} \rfloor \texttt{)} \rceil \end{array}}$$

```
      (diff-exp (exp1 exp2)
        (let ((val1 (value-of exp1 env))
              (val2 (value-of exp2 env)))
          (let ((num1 (expval->num val1))
                (num2 (expval->num val2)))
            (num-val
              (- num1 num2)))))
```

$$\boxed{\dfrac{\texttt{(value-of } exp_1 \ \rho\texttt{)} = val_1}{\begin{array}{l}\texttt{(value-of (zero?-exp } exp_1\texttt{) } \rho\texttt{)} \\ \quad = \begin{cases}\texttt{(bool-val \#t)} & \text{if } \texttt{(expval->num } val_1\texttt{)} = 0 \\ \texttt{(bool-val \#f)} & \text{if } \texttt{(expval->num } val_1\texttt{)} \neq 0\end{cases}\end{array}}}$$

```
      (zero?-exp (exp1)
        (let ((val1 (value-of exp1 env)))
          (let ((num1 (expval->num val1)))
            (if (zero? num1)
              (bool-val #t)
              (bool-val #f)))))
```

$$\boxed{\dfrac{\texttt{(value-of } exp_1 \ \rho\texttt{)} = val_1}{\begin{array}{l}\texttt{(value-of (if-exp } exp_1 \ exp_2 \ exp_3\texttt{) } \rho\texttt{)} \\ \quad = \begin{cases}\texttt{(value-of } exp_2 \ \rho\texttt{)} & \text{if } \texttt{(expval->bool } val_1\texttt{)} = \texttt{\#t} \\ \texttt{(value-of } exp_3 \ \rho\texttt{)} & \text{if } \texttt{(expval->bool } val_1\texttt{)} = \texttt{\#f}\end{cases}\end{array}}}$$

```
      (if-exp (exp1 exp2 exp3)
        (let ((val1 (value-of exp1 env)))
          (if (expval->bool val1)
            (value-of exp2 env)
            (value-of exp3 env))))
```

$$\boxed{\dfrac{\texttt{(value-of } exp_1 \ \rho\texttt{)} = val_1}{\begin{array}{l}\texttt{(value-of (let-exp } var \ exp_1 \ body\texttt{) } \rho\texttt{)} \\ \quad = \texttt{(value-of } body \ [var = val_1]\rho\texttt{)}\end{array}}}$$

```
      (let-exp (var exp1 body)
        (let ((val1 (value-of exp1 env)))
          (value-of body
            (extend-env var val1 env))))))
```