Principles of Computer Science

An Invigorating, Hands-On Approach

Joshua Crotts

A Logic Primer

What is logic?

- Logic is the use of *deductive reasoning* to analyze an *argument*.
- Arguments are comprised of *premises* and *conclusions*.
- *Premises* describe the reasoning of an argument.
- A *conclusion* is what follows from the premises.

Truth values and connectives

- *Propositions* are statements that are either true or false.
  - E.g., “The sky is blue”, “$2 + 2 = 5$”

- We assign *truth values*, i.e., “true” or “false”, to a proposition.

- *Connectives* allow us to modify the truth value of propositions and conjoin propositions.

- Five connectives in *zeroth-order logic*: $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$

Logical negation

- In most instances, for any proposition 'p', it is safe to use the phrase, "It is not the case that 'p' is true", to represent the *logical negation* of 'p'.

- Problem: sentences do not have a straightforward binary conversion between non-negation and negation.

- Truth table:

p	$\neg p$
$\top$	$\bot$
$\bot$	$\top$

Logical conjunction

- Represents the connection of propositions with non-symbolic words such as “and” and “but”.

- Both operands of a schema must be true for the logical conjunction to be true.

- Truth table:

p	q	$p \wedge q$
$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$
$\bot$	$\top$	$\bot$
$\bot$	$\bot$	$\bot$

Logical disjunction

- Represents the truth of at least one of two schema using phrases like “or”.
- Logical disjunction is inclusive-or.
- Truth table:

p	q	$p \vee q$
$\top$	$\top$	$\top$
$\top$	$\bot$	$\top$
$\bot$	$\top$	$\top$
$\bot$	$\bot$	$\bot$

Logical conditional

- Determines the truth conditions for a relationship between schema.
- When the antecedent is true and the consequent is false, the conditional is false.
- "*Implication* is the validity of the conditional".
- Truth table:

p	q	$p \rightarrow q$
$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$
$\bot$	$\top$	$\top$
$\bot$	$\bot$	$\top$

Logical biconditional

- True if both operands of the biconditional are the same.
- "*Equivalence*" is the validity of the biconditional".
- Truth table:

p	q	$p \leftrightarrow q$
$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$
$\bot$	$\top$	$\bot$
$\bot$	$\bot$	$\top$

Quantifiers

- In first-order logic we use quantifiers for one reason: as their name suggests, they quantify, or provide numeric amounts to, some entity.

- Universal quantifier:
  - To say that "All math majors are smart", we use *predicates* and variables: '$\forall x(M(x) \to S(x))$'
  - We say $S(x)$ means x is smart, and $M(x)$ represents x is a math major.

- Existential quantifier:
  - To say that "Some math majors are computer science majors", we use '$\exists x(M(x) \wedge C(x))$'.
  - We say $C(x)$ means x is a computer science major.

Identity

- Identity allows us to denote reference a particular entity.
- E.g., "Anyone who is the best computer science is Katherine Johnson":
  - '$\forall x(C(x) \rightarrow \forall y((C(y) \wedge B(x, y)) \rightarrow x = k))$'
  - We say $B(x, y)$ means x is better than y.
- Identity is a predicate; returns true or false if the identity relationship holds.

Set theory

- A *set* S is an unordered non-duplicate collection of values.
- An element x is in S means $x \in S$.
- The number of elements in a set S is denoted as $|S|$ also called the *cardinality*.
- A *subset* of S, namely S', is denoted as $S' \subseteq S$ if all elements of S' are elements of S.
- Two sets are *equivalent* if they are subsets of each other.

More about set theory

- The *union* of two sets S and T, i.e., $S \cup T$, is defined as the set of elements that are in either S or T or both.

- The *intersection* of two sets S and T, i.e., $S \cap T$, is defined as the set of elements that are in both S and T.

- The difference of two sets S and T, i.e., $S - T$, is defined as the set of elements that are in S but not in T.

- Some common mathematical sets: natural numbers $\mathbb{N}$, the integers $\mathbb{Z}$, the rationals $\mathbb{Q}$, the reals $\mathbb{R}$, and the complex numbers $\mathbb{C}$.

Functions

- *Functions* are maps between sets called the *domain* and the *range*.
- E.g., $f(x) = x + 5$ maps any number x to the set of x plus five. For instance, $f(5) = 10$.
- Substitute the function *parameters*, i.e., x, for the *arguments*, i.e., 5.
- We can write functions of multiple arguments:

$$\begin{aligned} g(x, y, z) &= 3x^2 + 4y + z \\ g(10, 2, 3) &= 3(10)^2 + 4(2) + 3 \\ &= 311 \end{aligned}$$

Recursive functions

- A *recursive* function is a function that calls itself.
- Think of addition: if we add two values n and m, we know that $n + 0 = n$. To solve $n + m$, we should solve $n + (m - 1)$. Then, we can propagate the result back up. Assume we know how to add and subtract one.
- E.g.,

$$\begin{aligned} add(3,4) &= 3 + 4 \\ &= 1 + (3 + 3) \\ &= 1 + (1 + (3 + 2)) \\ &= 1 + (1 + (1 + (3 + 1))) \\ &= 1 + (1 + (1 + (1 + (3 + 0)))) \\ &= 1 + 1 + 1 + 1 + 3 \end{aligned}$$

Data Structures

What are data structures?

- Data structures store data!
- Different ways of storing data for performance, space optimization, and so forth.
- Many are simple, some are wildly complex.

Arrays

- *Arrays* are contiguous blocks of storage where each block contains space for *n elements* of a given type.
- An advantage to using arrays are their quick access times.
- A disadvantage of arrays is that they are not resizable, their size must be known before creation.
- Arrays cannot store differing types; i.e., we can't store a string and an integer in the same array.

Array Lists

- Like arrays, *array lists* store elements of a type. Unlike arrays, they are resizable!

- Advantages:
  - Most implementations are quick to set up and understand, which leads to their widespread usage compared to other data structures.
  - As we said, they are resizable.
  - Insertion of new elements is easy.

- Disadvantages:
  - Easy to use, but not performant. Insertion and removal of elements is slow.

Linked Lists

- *Linked lists* are a series of *nodes*, or elements, linked together in a chain of sorts.
- Advantages:
    - Insertion, addition, and removal is quick! No need to resize/shift values.
- Disadvantages:
    - Element/index retrieval is slow; we no longer have contiguous elements in memory.

Stacks

- The *stack* data structure is a collection of elements that operate on the principle of last-in-first-out, or LIFO.
  - The last thing that we enter is the first thing removed.

- Advantages:
  - Fast insertion and removal operations via `push` and `pop`.

- Disadvantages:
  - Not as flexible as arrays or lists; cannot access arbitrary elements.

Queue

- The *queue* data structure is a collection of elements that operate on the principle of first-in-first-out, or FIFO.
  - The first thing that we enter is the first thing removed.
- Advantages:
  - Fast insertion and removal operations via `enqueue` and `dequeue`.
- Disadvantages:
  - Not as flexible as arrays or lists; cannot access arbitrary elements.

Sets

- *Sets* are similar to their mathematical counterpart; collection of unordered and non-duplicate elements.

- Advantages:
  - Easy to add and remove elements; we can also query the set for item presence.

- Disadvantages:
  - No ordering to values; no "indices" to elements of a set.

Maps

- *Maps* are association pairs/relationships. These pairs have a *key* and a corresponding value.

- Advantages:
  - Easy to determine whether a key exists in the map.
  - Trivial to setup a relationship between two values.

- Disadvantages:
  - No ordering to key/value pairs.

Trees

- *Trees* are like linked lists, but there are potentially multiple links to a node.
- Trees are recursive data structures because the elements of a tree are trees themselves.
  - E.g., binary trees are nodes with at most two children.
- Advantages:
  - Easy to describe relationships with real-world systems, e.g., mathematical structures, and even file systems.
- Disadvantages:
  - Hard to design, can become "left" or "right" leaning, decreasing performance.

Graphs

- A *graph* is a tuple $\langle V, E \rangle$, where V is the set of *vertices*, or nodes, and E is the set of *edges*.
- Edges are tuples, which serve as links between vertices.
- Edges can have a direction or be bidirectional.
- Edges in a graph may also be either *weighted* or *unweighted*, denoting a "cost".
- Advantages:
  - Applicable to lots of real-world concepts.
- Disadvantages:
  - Hard to write algorithms for, and can be costly in terms of performance and space.

Formal Languages

What are languages?

- To talk about languages, we first need to define an *alphabet*.

- *Alphabets* are sets, Σ, where each element is a distinct *symbol* or a grouping of symbols.

- A *language* L over an alphabet Σ is a subset of Σ where each element is an arrangement, or a permutation, of the alphabet.

Grammars

- *Grammars* describe the syntax of a language.

- We define a grammar G as a set of terminals T, a set of non-terminals T', and a set of production rules R.
  - A *terminal* is an atomic literal result of a production rule.
  - A *non-terminal* is a set of possible paths that a string can take in a production rule.
  - *Production rules* combine and define the relationship between terminals and non-terminals.

- (Extended) *Backus-Naur Form* grammars are a formalism to grammatical language constructions.

- Example of an BNF grammar for a prefix notation arithmetic expression language:

```
T       ::=   "0" | "1" | ...  | "9" | "+" | "-" | "*" | "/"
T'      ::=   R* WS NUM OP EXPR
WS      ::=   " "
NUM     ::=   ("0" | "1" | ...  | "9")+
OP      ::=   "+" | "-" | "*" | "/"
EXPR    ::=   "(" OP WS EXPR WS EXPR ")"
         |    NUM
R*      ::=   EXPR
```

Finite automata

- *Finite automata* are, in essence, very weak computers, or models of computation.
  - They describe *transitions* between *states* in some model.
  - Use input symbols belonging to an alphabet Σ.

- A deterministic finite automaton F is a quintuple $\langle Q, \Sigma, \delta, q_0, F\rangle$.
  - Q is the set of states.
  - δ is a transition function.
  - q_0 is the start state.
  - F is the set of accepting states.

Regular languages

- *Regular languages* are languages recognized by a deterministic finite automaton.

- Any DFA can be converted into a regular expression and vice versa.

- See the book for details on the syntax.

Lexical analysis

- *Lexical analysis* involves assigning meaning to sequences of characters.

- Example: in a string containing "$1 + 23 \cdot 41$", we might *tokenize* these lexemes by assigning the token **Number** to the lexemes '1', '23', and '41'.

- We use lexical analysis primarily when designing the grammar of a programming language.

Syntactic analysis

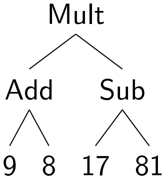
- *Syntactic analysis*, also called *parsing*, is determining whether a sequence of tokens conform to a language grammar.

- When parsing tokens, we build data structures called parse trees, which are then converted into abstract syntax trees.

- *Parse trees* are hierarchical representations of tokens.

Abstract syntax trees

- Whereas parse trees describe the syntactic structure of an input, *abstract syntax trees* explains the relationships between subtrees.
- Abstract syntax trees strip extraneous characters such as separators that do not contribute to a node in the tree.
- Example: AST of ‘$((9\ 8\ +)\ (17\ 81\ -)\ \cdot)$’:



λ-calculus

- The λ-calculus in the early 1930s is an abstract machine for modeling computation.

- We have variables, $x, y, ..., z$, function definitions/abstractions $\lambda v.B$ where v is a variable and B is a λ-calculus term, and function application $(M\ N)$ where M and N are λ-calculus terms.

- Seems limited at first glance, but we can represent many computations and programs with the λ-calculus.

- Very, very, very slow from a performance standpoint, but that wasn't Alonzo Church's point!

Programming and Design

What language for our language?

- To explore concepts in programming languages and computer science, we need to actually start programming!
- We will develop our *own* programming language in due time.
- Until then, we need to get familiar with C: the language of choice.
- Why C?
  - It's small
  - It's fast
  - Tried and tested (to some degree)

“Hello, world!” in C

- Refer to the book for a more in-depth explanation.

```c
#include <stdio.h>

int main(void) {
  printf("Hello, world!\n");
  return 0;
}
```

- Example of addition:

```c
#include <stdio.h>

int add(int n, int m) {
  if (m == 0) {
    return n;
  } else {
    return 1 + add(n, m - 1);
  }
}

int main(void) {
  printf("%d\n", add(3, 4));
  return 0;
}
```

Conditionals

- Conditionals allow us to make decisions in our program.
- Change control flow.
- The *conditional expressions* must resolve to either `true` or `false`.

```
int main(void) {
  int x = 0;
  if (someCondition) {
    x = 5;
  } else if (someOtherCondition) {
    x = 10;
  } else {
    x = -1;
  }
  return 0;
}
```

Pointers

- Passing values as arguments to functions is by *value*.
- Modifying that value inside the function does not change its value on the outside.

```
void swap(int x, int y) {
  int tmp = x;
  x = y;
  y = tmp;
}
```

- Pointers are locations in memory.
- We can use them to pass a reference to the variables we want to update inside the function.

```
void swap(int *x, int *y) {
  int tmp = *x;
  *x = *y;
  *y = tmp;
}
```

Arrays

- Arrays, of course, are fixed-sized data structures.
- Size must be known at compile-time.
- Indices are indexed from zero.
- If we don't know the size at compile-time, use `malloc`.

```
int main(void) {
  int[] arr = new int[5];
  arr[0] = 5;
  arr[1] = 10;
  arr[2] = 20;
  arr[3] = 40;
  arr[4] = 45;
  return 0;
}
```

Strings

- Strings are nothing more than an array of characters.
- String literals are immutable.

```
int main(void) {
  const char *s1 = "Hello, world!";
  char[] s2 = "Hello, world!";
  s2[5] = '?';
  return 0;
}
```

Loops (1)

- While loops are for repeating a task an indeterminate number of times.
- Example: Collatz conjecture.

```
int main(void) {
  int n = ...;
  int i = 0;
  while (n != 1) {
    if (n % 2 == 0) {
      n = n / 2;
    } else {
      n = 3 * n + 1;
    }
    i++;
  }
  return 0;
}
```

Loops (2)

- For loops are used when we want to repeat a task a determinate number of times.
- Example: computing factorial of n.

```
int main(void) {
  int n = ...;
  int res = 1;
  for (int i = 1; i <= n; i++) {
    res *= i;
  }
  return 0;
}
```

Structs

- Structs allow us to group data to make an "object" of sorts.
- Example: consider a `student` struct.

```
struct student {
  char id[256];
  double gpa;
};

int main(void) {
  struct student s1;
  strcpy(s1.id, "Katherine");
  s1.gpa = 4.0;

  struct student *s2 = malloc(sizeof(student));
  strcpy(s2->id, "Bjarne");
  s1.gpa = 3.5;
  return 0;
}
```

Unions

- Unions let you store multiple types of values under one “umbrella”.

```
union data {
  int number;
  char ch;
  char *string;
  bool val;
}

int main(void) {
  union data v1, v2, v3;
  v1.number = 5;
  v2.ch = 'A';
  v3.val = false;
  return 0;
}
```

$\mathcal{L}_{\mathsf{PF1}}$: A prefix arithmetic language

- To start things small, we will interpret a prefixed arithmetic language.

```
expr          ::=  application
                |  datum
                |  comment
application   ::=  '(' expr* ')'
                |  '[' expr* ']'
                |  '{' expr* '}'
datum         ::=  number
                |  symbol
comment       ::=  ';' (.  - '\n')*
number        ::=  ('+'|'-')?  (digit)+ ('.'  (digit) *)?
symbol        ::=  symchar (symchar | number)*
pf1           ::=  expr+
```

Representation independence with respect to ASTs

- Our programming languages will make use of Daniel Holden's *mpc* library, specifically for generating ASTs.
- **Problem:** what if we want to swap this library out in the future?
- **Solution:** write functions that tap into the library and use *these* functions in our interpreter.
- We will revisit representation independence multiple times.

$\mathcal{L}_{\mathrm{PF2}}$: Now with environments!

- A programming language without variables is pretty lame.

- We need to introduce the notion of *environments*.

- An environment binds identifiers to their values. E.g.,

  ```
  (define x 5)
  (define y 6)
  ```

  We define the association $x \mapsto 5$ and $y \mapsto 6$.

Interpretation

$\mathcal{L}_{\text{COND}}$: Conditionals and Decisions

- Conditionals, as we saw in our C primer, allow us to divert program control based on decisions.

- To ease our transition, we first introduce a language with only booleans, then boolean expressions, then conditional expressions.

```
expr          ::=  application | ...
application   ::=  cond | if | ...
cond          ::=  ‘(cond’ cond-clause* else-clause ‘)’
cond-clause   ::=  ‘[’ expr ‘ ’ expr ‘]’
else-clause   ::=  ‘[’ ‘else’ ‘ ’ expr ‘]’
if            ::=  ‘(if ’ expr ‘ ’ expr ‘ ’ expr‘)’
```

$\mathcal{L}_{\text{LOCAL}}$: Local identifiers and values

- Our language is *lexically-scoped*, meaning identifiers obtain their values by when they were declared.

- Introduces `let`, `let*` bindings.

```
expr          ::=  application | ...
application   ::=  let | letstar | ...
let           ::=  'let (' let-bndg+ ')' expr
letstar       ::=  'let* (' let-bndg+ ')' expr
let-bndg      ::=  id ' ' expr
```

$\mathcal{L}_{\text{PROC1}}$ & $\mathcal{L}_{\text{PROC2}}$: Recursive procedures

- Functions, or procedures, define a callable section of code with or without parameters.

- Their definition comes through `lambda`, which means we can define *anonymous* and *non-anonymous* functions.

```
expr         ::=  application | ...
application  ::=  proc | ...
proc         ::=  'lambda' '(' id* ')' expr
```

$\mathcal{L}_{\text{LETREC}}$: One more time with `letrec`

- Sometimes, we do not want to expose a function definition into the *global namespace*.
- **Solution:** we can define functions inside a `let` or `let*` block.
- **Problem:** these functions cannot be recursive.
- **Solution:** use `letrec`!

Different datatypes

- Restricting ourselves to working with only integers, booleans, and functions is unnecessary.

- We provide descriptions for three languages: $\mathcal{L}_{\text{CHAR}}$, $\mathcal{L}_{\text{STRING}}$, and $\mathcal{L}_{\text{EQUAL}}$.
  - $\mathcal{L}_{\text{CHAR}}$ describes operations for working with single characters.
  - $\mathcal{L}_{\text{STRING}}$ allows us to create and manipulate strings.
  - $\mathcal{L}_{\text{EQUAL}}$ defines predicates for determining equality amongst values.

Functional Programming

$\mathcal{L}_{\text{QUOTE}}$: Quoted expressions

- How can we turn code into data?
- Quoting!
- `'(+ 2 3)` resolves to `(+ 2 3)`.
- What might this lead us towards?

$\mathcal{L}_{\text{LIST}}$: Pairs and lists

- We need some type of data structure.
- *Pairs* contain a *first* and a *rest*.
- We create pairs using `cons`, and reference the elements using `first` and `rest`.
- `first` returns the first item of the pair.
- `rest` returns the second item of the pair, or the rest of the list if called on a list.

$\mathcal{L}_{\text{QUASI}}$: Quasiquotes

- Quoted data is fun, but what does this evaluate to?

```
(define x 5)
(define y 6)
'(10 30 x 50 60 y)
```

`'(10 30 x 50 60 y)`... would it not be more sensible to resolve the x and y?

- Quasiquoting and unquoting allows us to do this!

```
`(10 30 ,x 50 60 ,y)
```

$\mathcal{L}_{\text{VARIADIC}}$: Support for variadic-argument functions

- A function that is defined to receive any number of arguments is called *variadic*.

- Under the hood, we translate these into a list of arguments.

- The function processes these arguments as if they were received a list of values.

First-class & Higher-order functions

- In our language and other functional programming languages, functions are *first-class citizens*, meaning they can be passed around as arguments to functions and returned from functions.

- Example:

```
(define compute-bill
  (λ (tip-pt)
    (λ (tax-pt)
      (λ (sub serv)
        (let ([tax-amt
               (+ sub
                 (* sub
                   (/ tip-pt 100)))])
          (+ (+ tax-amt
               (* (/ tax-pt 100)
                  tax-amt))
             serv))))))
```

$\mathcal{L}_{\text{EVAL}}$: Evaluation and application

- We have a way of converting code into data via quoting, but what about the other way around?
- Two new forms: `eval` and `apply`.
- `eval` receives a quoted expression, or data, and attempts to evaluate it. E.g., `(eval '(+ 2 3))` resolves to 5.
- `apply` applies a function to a list of arguments. E.g., `(apply cons '(2 3))` resolves to `(2 . 3)`.

Accumulator-passing style

- *Accumulators* are values that we construct when a function is in *tail-position*.
- A function call is in tail position if it is the last action performed before a "return".
- We accumulate the result in a parameter, hence the term "accumulator-passing style".

Continuation-passing style

- A *continuation* is, in effect, "the rest of a computation".
- We use continuations to direct program control to where **we** want it to go next.
- E.g., *k* is the continuation!

```
(define fact-cps
  (λ (n k)
    (cond
      [(zero? n) (k 1)]
      [else
        (fact-cps (sub1 n)
                  (λ (v)
                    (k (* n v))))])))
```

- We invoke this by `(fact-cps 5 (λ (v) v))`

Nested interpreters

- Our language is now powerful enough to where we can write interpreters from within the interpreter! We call this *nested interpretation*.

- For nested languages, we need to define *recognizer* functions and *reducer* functions.
  - Recognizer functions determine whether a value represents some structure.
  - Reducer functions evaluate the structured data.

- Tons and tons of examples in the book.

Imperative Programming

$\mathcal{L}_{\mathsf{SET}}$: Assignment statements

- C allows us to reassign variables after their initialization.
- Until now, our language does not let us.
- Doing so raises questions about the purity of our language.

```
expr         ::=  application | ...
application  ::=  set | setfirst | setrest | ...
set          ::=  'set! ' symbol expr
setfirst     ::=  'set-first! ' symbol expr
setrest      ::=  'set-rest! ' symbol expr
```

$\mathcal{L}_{\text{BEGIN}}$: Sequential expressions

- Assignment statements, e.g., `set!`, do not return a value.

- Therefore, we should add a construct that allows us to chain statements and expressions in a sequence.

- How does this help us? *Closures* are now easier to visualize.

```
expr          ::=  application | ...
application   ::=  begin | ...
begin         ::=  'begin ' expr+
```

$\mathcal{L}_{\text{OUT}}$: Fancier output

- In C we use `printf` for formatted output. We can output strings, booleans, integers, whatever we wish.

```
expr          ::=  application | ...
application   ::=  printf | ...
printf        ::=  'printf' expr expr*
```

Parameter-passing styles

- *Pass-by-value:* pass a copy of each argument to functions.
- *Pass-by-reference:* pass a memory reference of each argument to functions. Mutating a value in the function modifies the value outside as well.
- *Lazy evaluation by name:* evaluate arguments only as they are referenced in the body of a function.
- *Lazy evaluation by need:* evaluate arguments only as they are referenced in the body of a function, but save the result of the expression to avoid recomputation.

- Pairs and lists are dynamic data structures; i.e., they are resizable.
- *Vectors* are like C arrays; they cannot be resized after their declaration, but provide constant lookup times.

```
expr         ::=  application | ...
application  ::=  vector
              |   vector-set
              |   vector-get
              |   ...
vector       ::=  'make-vector' expr
vector-set   ::=  'vector-set!'  id expr expr
vector-get   ::=  'vector-get' id expr
```

$\mathcal{L}_{\text{LIB}}$: External libraries

- Libraries, or auxiliary files with function definitions, prevent the need to constantly rewrite functions.

- Requires careful parsing; how do we handle circular dependencies or duplicate function definitions?

```
expr         ::=  application | ...
application  ::=  include | ...
include      ::=  'include ' string
```

$\mathcal{L}_{\text{BIGNUM}}$: Arbitrarily-precise numbers

- Using only 64-bit `double` numbers limits our program capabilities. What if we want to work with arbitrarily large values?

- No new language features aside from reworking our s-value for numbers to use *gmp* and *mpfr*.

- To simplify successive discussions, we will not use $\mathcal{L}_{\text{BIGNUM}}$ following this section.

$\mathcal{L}_{\mathsf{IN}}$: Improved user input

- In C we can use `getline` and `fgets` to read strings in from different sources.
- We then parse these using `sscanf` or some other roughly-equivalent function.
- $\mathcal{L}_{\mathsf{IN}}$ adds `read-string` and `read-number` for reading strings and numbers, respectively, from standard input.

$\mathcal{L}_{\text{FILE I/O}}$: File input and output

- Working with files is a prominent part of programming and software development.

- In C we use `FILE` and auxiliary functions to read data from files.

- $\mathcal{L}_{\text{FILE I/O}}$ uses the C primitives to add support for reading from and writing to files.

$\mathcal{L}_{\text{LOOP}}$: An iterative approach to problem-solving

- Recursion is a great and powerful concept, but we can very easily overflow the procedure call stack.
- Moreover, some concepts are harder to understand when the only tool at our disposal is recursion.
- $\mathcal{L}_{\text{LOOP}}$ adds a `do` loop construct, which functions identically to a `while` loop in C.

```
expr         ::=  application | ...
application  ::=  do | ...
do           ::=  'do ' expr expr
```

$\mathcal{L}_{\text{MACRO}}$: A simple macro system

- *Macros* are textual substitutions in code.
- We use the preprocessor in C, but we do not have such a thing in our language.
- What do macros give us? Lots of helpful language constructs that are otherwise impossible or cumbersome, e.g., *promises*.

```
expr          ::=  application | ...
application   ::=  macro | ...
macro         ::=  'define-macro (' id id* ')' expr
```

Compilation

An assembly primer

- We will write a small compiler for our language.

- Recall that compilers, in general, target machine-dependent assembly language; we will choose x86/64 assembly.

- Compilers are *much* faster than interpreters, hence the desire!

- Assembly is mnemonic-driven; small instructions to do small tasks. We operate primarily on *registers*: 64-bit slots for values on the CPU.
  - `movq %rax, %rbx` moves the data from register `%rax` into register `%rbx`.

Compiling $\mathcal{L}^{-}_{\mathsf{PF1}}$ to $\mathcal{L}^{-}_{\mathsf{PF1}_{x64}}$

- Our first language supports printing only constant integer values.

```
expr       ::=   '(call (print' ' ' constant '))'
constant   ::=   [0-9]+
pf1-       ::=   expr*
```

- After this we expand out to include simple binary operations and expressions.

```
expr        ::=   '(call (print' ' ' (constant | arithexpr) '))'
arithexpr   ::=   '(' binop ' ' constant ' ' constant ')'
binop       ::=   '+' | '-' | '*' | '/'
constant    ::=   [0-9]+
pf1         ::=   expr*
```

Compiling $\mathcal{L}_{\mathsf{PF2}}$ to $\mathcal{L}_{\mathsf{PF2}_{x64}}$

- We want to support variables; let's add those! All variables are allocated on the stack. Inefficient, but simple.

```
expr       ::=  call
             |  var
             |  arithexpr
             |  constant
             |  id
call       ::=  '(call (print' ' ' expr '))'
var        ::=  '(var ' id ' = ' expr')'
arithexpr  ::=  '(' binop binopval binopval ')'
binopval   ::=  {call | constant | id};
id         ::=  [a-zA-Z]+
pf2        ::=  expr*
```

Compiling $\mathcal{L}^{-}_{\text{COND}}$ to $\mathcal{L}^{-}_{\text{COND}_{x64}}$

- Before we compile conditionals, we should get boolean expressions to work.

```
expr        ::=   cmp-expr | ...
cmp-expr    ::=   ‘(’ cmp-op ‘ ’ expr ‘ ’ expr ‘)’
cmp-op      ::=   ‘?=’ | ‘!=’ | ‘<’ | ‘<=’ | ‘>’ | ‘>=’
cond-       ::=   expr*
```

- After this we can add an `if` statement.

```
expr    ::=   if | ...
if      ::=   ‘(if ’ expr expr expr ‘)’
cond    ::=   expr*
```

Compiling $\mathcal{L}^{+}_{\text{COND}}$ to $\mathcal{L}^{+}_{\text{COND}_{x64}}$

- Programming languages aren't very powerful without some way to repeat an action.

- Since we do not yet have procedures, we cannot implement recursion.

```
expr       ::=  while | ...
while      ::=  '(while ' expr ' ' expr ')'
condplus   ::=  expr*
```

Compiling $\mathcal{L}^{-}_{\mathsf{PROC}}$ to $\mathcal{L}^{-}_{\mathsf{PROC}_{x64}}$

- On the journey to functions, we will first implement *subroutines*: or functions that do not receive nor return values.

```
expr    ::=  proc | ...
proc    ::=  '(proc ' id ' ' '(' id* ')' lstmt ')'
lstmt   ::=  expr lstmt | expr
id      ::=  [a-zA-Z]+
proc-   ::=  expr*
```

Compiling $\mathcal{L}_{\mathsf{PROC}}$ to $\mathcal{L}_{\mathsf{PROC}_{x64}}$

- Subroutines are boring!

```
expr      ::=  call | proc
call      ::=  ‘(call ’ id ‘(’ expr* ‘))’
procdecl  ::=  ‘(proc ’ id ‘(’ id* ‘)’ expr* ‘)’
proc      ::=  expr+
```

Compiling $\mathcal{L}^{+}_{\mathrm{PROC}}$ to $\mathcal{L}^{+}_{\mathrm{PROC}_{x64}}$

- Some functions do not compile correctly in $\mathcal{L}_{\mathrm{PROC}_{x64}}$. We need to fix them!

- **Problem:** we delay setting argument registers until after all arguments are evaluated.

- **Solution:** evaluate the arguments to a function *in reverse*, push the result to the stack via `pushq`. Then, once all arguments have been evaluated, pop the results off the stack into the appropriate argument-registers via `popq`.

- We need a data structure to make this a truly powerful language! Let's implement stack-allocated arrays.

```
expr        ::=  getindex | setindex | ...
decl        ::=  arraydecl | ...
arraydecl   ::=  '(array ' number ')'
getindex    ::=  '(get-index ' id ' ' expr ')'
setindex    ::=  '(set-index ' id ' ' expr ' ' expr ')'
array       ::=  decl* expr*
```

Compiling $\mathcal{L}_{\text{FLOAT}}$ to $\mathcal{L}_{\text{FLOAT}_{x64}}$

- We already store integers in registers; can we not do the same for floating-point values?
- Answer: **no!** Floating-point values are considerably more difficult to tackle.
- We cannot use local variables; everything is declared in the data segment.

```
expr      ::=  arithexpr | setexpr | callexpr
callexpr  ::=  '(call ' id id* ')'
proc      ::=  '(proc main ()' '(' expr+ '))'
constant  ::=  number
float     ::=  vardecl* proc
```

Memory Management

Stack-allocated (static) memory

- The *stack* is a small section of memory for local variable declarations (not using `malloc` or its derivatives).

- We also use the stack for function calls, i.e., function arguments, return values, and so forth are stored in *activation records*.

- When a function returns, its activation record is removed from the stack, thereby removing all stack-declared variables.

Heap-allocated (dynamic) memory

- The *heap* is a collection of blocks that our program can “tap into” when allocating memory at runtime.
- We have seen this with functions, e.g., `malloc`, `calloc`, `realloc`, `strdup`, and so forth.
- In C, we have to `free` this memory, otherwise we cause a memory leak.

Garbage collection

- Scheme is particularly tricky to allocate/deallocate memory for, because the lifetime of a function/variable is not always unknown.
- Deallocating at the wrong time will cause an undefined variable reference or crash the interpreter.
- A *garbage collector* keeps track of “live” heap references and deallocates these chunks when nothing points to them (i.e., they are no longer live).

Garbage collection (cont)

- We write two garbage collectors in the book: a simple one and a reference-counted garbage collector.

- The simple garbage collector simply keeps track of the allocations made and frees them before ending the program.
    - Incredibly simple, but not very useful.

- The *reference-counted garbage collector* counts each pointer to an object in memory; once that number reaches zero for an object, it is no longer reachable and its memory is freed.

Event-Driven Programming

Concurrent programming

- Our programs so far have been loaded in via files. What if we want to run a program and make changes on the fly?
  - We can implement a *read-evaluate-print-loop*.

- **Problem:** our system has to constantly listen for input and be ready to receive it, so how can the system also evaluate expressions in the interpreter?

- **Solution:** multithreading!

Threading

- *Threads* manage separated sequence of actions for the current program to execute.

- **Problem:** multithreading opens the nasty can of worms that contains data races/race conditions. Race conditions are "competitions" for a piece of data; one thread might write into a value using an old value and another thread can then use an incorrect value.

- **Solution:** mutexes and condition variables!

Multithreading and garbage collection

- Previous versions of our garbage collectors were “stop the world” garbage collectors, i.e., interpretation stops to wait for the collector to finish.

- Stop the world garbage collectors are slow!

- We can integrate multithreading into the mix and use a separate thread for our reference-counted garbage collector.