Implementing an Interpreter in C++

Petr Ročkai

Organisation

- theory: ~50 minutes every two weeks
- coding: all the remaining time

Assignments

- 2 weeks → 1 topic → 1 assignment
- you should get most of the work done during the seminar
- assignments include writing tests!
- on my desk (in email, git, ...) by 8am on even Wednesdays

Grading

- you pass if you implement a game of tic-tac-toe
  running in the interpreter you implemented

Your Own Programming Language (in 6 easy steps)

- Lexers and Parsers
- Symbol Tables
- Evaluating Expressions
- Type Checking (*)
- Memory Management (*)
- Talking to the Outside World

Organisation (cont'd)

- seminar attendance is optional
- you may skip starred topics if you have trouble keeping up
- but only if you attended 5 seminars per topic skipped

What You Need To Know

- we will use C++ 11 (or better)
- version control of your choice
- UNIX strongly recommended
- write automated tests (eg. shell scripts)

Part 1: Lexers and Parsers

Lexical Structure

- the source code is ASCII (Unicode) text
- working one character at a time is not fun
- lexer converts text into a stream of tokens

Token Categories

- keywords
- identifiers
- literals: strings and numbers
- operators
- brackets

Lexer is a Finite State Automaton

- the token structure is regular
- example: an identifier is `[a-zA-Z][a-zA-Z0-9]*`
- another one: a number is `[0-9][0-9]*`
- needs to deal with whitespace between tokens, too

Lexer

- reads characters from the input file
- outputs tokens for future processing

Token

- represented by a data type
- remembers the text and category
- also where it came from

```
struct Token
{
    std::string text;
    int lineno;
    enum Cat { If, Else, Endif,
               Identifier, ParenOpen, ParenClose,
               LitString, LitNumber } cat;
};
```

```cpp
struct Lexer
{
    Lexer( const char *filename );
    Token next(); /* main interface */

protected:
    std::ifstream in;
    std::string buf;

    /* state machine */
    Token identifier();
    Token literal();
    /* ... */
};
```

The next function reads and returns the next token

```
Token Lexer::next()
{
    whitespace();
    buf += ( c = in.get() );
    if ( std::isalpha( c ) )
        return identifier();
    switch ( c )
    {
        case '=': // ...
    }
}
```

The state machine could be implemented by using one method for each automaton state:

```
Token Lexer::identifer()
{
    char c;
    while ( std::isalnum( c = in.get() ) )
        buf += c;
    in.unget();
    if ( is_keyword( buf ) ) return // ...
    return Token( Token::Identifier, buf, ... );
}
```

`next()` from previous slide is the initial state

Parsers

- typically a context-free language
- terminals (symbols) are the tokens
- a stack (or recursion) is required for parsing
- selection of algorithms (LL, LR, GLR, monadic, rec. descent)
- different trade-offs

Context

- parser reads tokens from the lexer
- creates an Abstract Syntax Tree (AST for short)

Expressions: Prefix (eg. LISP)

- easy to parse, hard-ish to read, annoying to write
- variadic operators
- `(+ 1 2 (* 3 4) 5)`

Postfix (eg. PostScript)

- very easy to parse, hard to read, easy-ish to write
- unambiguous even without parens
- `3 4 * 1 + 2 + 5 +`

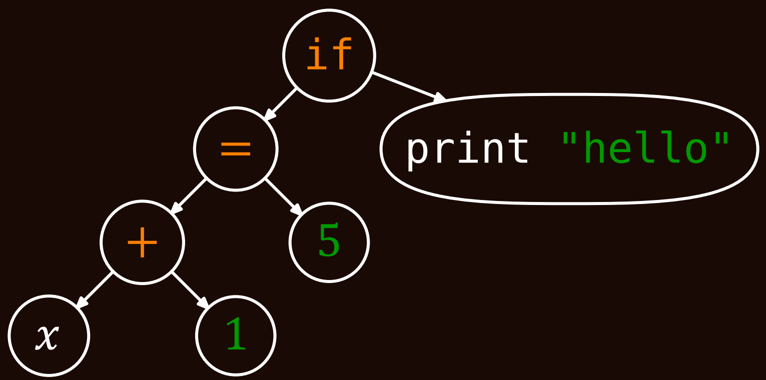
Infix (eg. everybody else)

- hard to parse, easy to read, easy-ish to write
- `1 + 2 + 3 * 4 + 5`

Abstract Syntax Tree

- internal representation of the source code
- tree with different node types

```
if ( x + 1 = 5 ) print "hello"
```



- reflects the structure of the (context-free) grammar

AST in C++

- use `std::shared_ptr` for children
- don’t overdo it (many things can be kept as values)

```
template< typename T >
using Ptr = std::shared_ptr< T >;

struct Expression { /* ... */ };
struct Statement { /* ... */ };
struct If : Node {
    Ptr< Expression > condition;
    Ptr< Statement > body;
};
```

AST: Representing Alternatives

- you can use `std::variant` (since C++17)
- or use `Union` from brick-types (see study materials in IS)
- or use enums and write out switches by hand (eww)

```
struct Expression;
using Atom = Union< Identifier, Literal >;
struct Binary { Ptr< Expression > left, right; };
using ExpBase = Union< Atom, Binary, Unary > {};
struct Expression : ExpBase { using ExpBase::Union; };
using Statement = Union< Expression, Block, If >;
```

Parsing: Context-Free Languages

- grammars with one-to-some rules
- alternatively: stack machines
- the goal is reconstructing a grammar derivation
- grammars are often ambiguous
- subsets of CFLs: LL(1), LR(1), LALR(1), …
- can be parsed more efficiently
- eg. limited lookahead, no or limited backtracking

Parsing: Recursive Descent

- parse LL(k) languages in linear time
- easy to write in direct C or C++
- no fancy generators needed (ie. no `yacc` nor `bison`)

Method

- look at one token and the grammar rules
- find which rules could have produced this token
- if there's only one, you know which rule to pursue
- otherwise the grammar is not LL(1)
- you can try looking at two tokens instead: LL(2)

Recursive Descent in C++

```
struct Parser
{
    Lexer lexer;
    Token tok;

    Toplevel toplevel();
    Call call();
    Identifier identifier();
    Ptr< Expression > expression();
};
```

- each non-terminal gets a function (more or less)
- each function returns the corresponding AST node

```
Ptr< Expression > Parser::expression()
{
    if ( tok.cat == Token::Identifier )
        return make_expr( identifier() );
    /* ... */
    if ( tok.cat != ParenOpen )
        fail( "opening paren" );
    shift();
    if ( tok.cat == Token::Identifier )
        return make_expr( call() );
}
```

- looks a bit like the lexer
- `shift()` grabs the next token into `tok`

Parsing: Reporting Errors

- LL(1) parsers can easily give nice error messages
- what you found vs what you expected to find
- the `Token` remembers where it came from

Example:

- parse error at `[LitString "bar" at line 3]`,
- expected an operator, identifier, `if`, `while` or `let`

Assignment (weeks 1 & 2)

- come up with decent syntax (could be LISP-like)
- conditionals, loops, expressions, variables & functions
- create corresponding AST for your language
- write a lexer and a parser to generate the AST
- write a pretty-printer for the AST
- write a dozen or so small example programs
- add a script to check that parse + prettyprint is idempotent

Due 8th of March, 8am!

Assignment Hints

- use prefix expressions (saves a lot of time)
- straight LISP-like syntax is LL(1)
- think about the grammar before writing too much code
- think about what you need in a programming language
- don't forget about local variables
- use C++ facilities (vectors, maps, sets) whenever useful
- don't lose much sleep over parsing speed
- you can find inspiration in `ex-parser.tar.gz` in the IS

Part 2: Symbol Tables

Lexical Scoping

- this is the contemporary norm
- alternative: dynamic scope (shell, elisp)
- alternative: no local variables

Symbol Tables

- keep track of what is in scope
- offer efficient lookup of definitions
- possibly also keep track of values

From Identifiers to Integers

- string comparison is slow
- the set of identifiers in a program is static
- we can assign a unique number to each identifier

For example:

- put all identifiers in a hashset or a search tree
- assign numbers in iteration order
- build a number → identifier (string) map

Lexical Scopes

- the global scope is shared by everything
- scopes can be nested

```
int global;
void foo()
{
    int local;
    if ( int z = local + global )
        printf( "z is not zero: %d\n", z );
    /* z no longer defined here */
}
/* 'local' is no longer defined here */
```

- scope nesting is rigid and does not change at runtime

Lexical Scoping: Implementation

- every lexical scope gets a (static) symbol table
- symbol tables get references to their parents
- if a symbol is not found, the table asks its parent scope

```
int Scope::lookup( int id )
{
    if ( idmap.find( id ) == idmap.end() )
        return parent.lookup( id );
    /* ... */
}
```

Static Checks

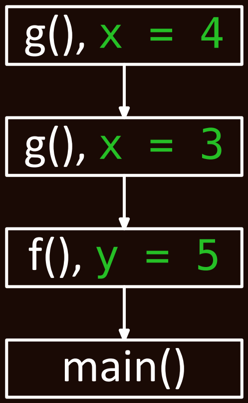
- correct syntax does not mean the program is well-formed
- variables must be defined before they are used
- functions must be defined before they are called
- (we will deal with type checking later)

- symbol tables are how these checks are done

Execution Stack

- functions call other functions (or themselves)
- the interpreter needs to keep track of this
- may consist of pointers to AST nodes
- if variables are mutable, keeps track of their values

```
void g( int x )
{
    g( x + 1 );
}
void f() { int y; g( 3 ); }
int main() { f(); }
```



Mutable Variables

- each activation record needs a copy of the value
  - activation record = stack frame
- option one: index stack frames by identifiers
  - less efficient, easier to implement
- option two: pre-compute a fixed layout for frames
  - store variable offsets in the static symbol table
  - more efficient but more work to implement

Dynamic Scope

- in lexical scoping, the parent is the enclosing block
- if the scope parent is the caller, you get dynamic scoping
- the scope lookup proceeds along the execution stack
- sometimes quite powerful, usually very confusing

Examples

- shell variables
- `perl` optionally (only some variables)
- old LISPs (including emacs lisp)
- Common Lisp optionally

Lexical Closures

- you may want to allow local function definitions
- a bit like C++ lambda expressions
- capture the lexical scope at the point of definition
- carry the scope (symbol table) around

```
void f( std::vector< int > &vec )
{
    std::for_each( vec.begin(), vec.end(), [&]( int x )
    { std::cout << vec.front() - x << std::endl; } );
}
```

Lexical Closures: Lifetime

- C++ lambdas capture by name or by reference
- if a reference-captured value goes out of scope, SIGSEGV
- in "dynamic" languages, this is usually different
  - reference-captured values live as long as needed
  - even if their original scope is gone
  - you need a garbage collector to do this
- capture by reference is usually more useful
  - in imperative languages, that is

Walking the AST

- use recursion to visit children of a node
- use type-based matching from `Union` where appropriate

```
expr.match(
    [&]( IfLike  &stmt ) { recurse( stmt.condition ); },
    [&]( DefLike &stmt ) { recurse( stmt.body ); },
    /* ... */ );
```

- first pass builds the symbol tables
- second pass checks that all identifier uses are correct

Symbol Tables: Summary

- static table for each lexical unit (function, block)
  - ensure functions and variables are in scope when used
  - possibly store auxiliary data (frame offsets)
- values are stored somewhere else (execution stack)
  - can use `std::map` from identifiers to values

Assignment (weeks 3 & 4)

- design and implement a symbol table data structure
- implement string → integer key mapping for identifiers
- write code to build all symbol tables from the AST
- check that all variables are in scope when used
  - print an error message otherwise
- figure out how to store values (at least integers and strings)
- write tests for everything above

Assignment Hints

- don't forget to use `std::map` and/or `std::unordered_map`
- take advantage of pattern matching in `Union`
- you can print symbol tables and use text comparison again
- try attaching local symbol tables to AST nodes

- ideally, a symbol table applies to one node + all its children
- sorry, no code hints this time, you did too well on parsers :-)

Part 3: Evaluating Expressions

Overview

- values and variables
- evaluation order
- recursive evaluators
- RPN evaluators

Evaluator

- an expression evaluator is the heart of an interpreter

Roles

- arithmetic and other elementary operations
- variable lookup
- function calls and argument passing
- control flow

Representing Values

- easy if all you have is integers
- otherwise, disjoint unions could work
- also useful for run-time type checking

Alternative (advanced)

- raw data (C unions) with type erasure
- needs a solid static type system

Alternative

- objects (as in OOP)

From Symbols to Values

- expressions without variables are boring
- symbol tables to the rescue

L-values and R-values

- ordinary variable use is R-value use
- a variable reference is replaced by its current value
- does not work for assignment (mutable variables)
- L-value stands for the address of a variable

Evaluation Order

- relevant for function application (calls)
- also for built-ins (control flow)

Normal

- expand the body first
- substitute un-evaluated arguments
- also known/implemented as: call by name, lazy

Strict

- compute argument values first
- also known/implemented as: call by value, eager

(Mostly) Imperative Programming

- call by value
- call by name (thunks)
- call by reference (pointers)
- call by object reference (call by sharing)
- call by value result (by value return)
- call by need (lazy)
- call by macro expansion (text-based)
- call by future (concurrent)

Thunks

```
int f() { std::cerr << "!"; return 3; }
int strict( int value ) { return value + value; }
int normal( std::function< int() > value )
{
    return value() + value();
}

int main()
{
    std::cerr << strict( 3 + f() );
    std::cerr << normal( []{ return 3 + f(); } );
}
```

Evaluation Order in C++

- almost all expressions are eager
- logical operators are lazy / “short circuiting”
- statements (`if`) are “lazy”
- promise/futures for lazy evaluation
- `std::future` / `std::async` with `std::launch::deferred`
- basically an explicit, type-safe thunk

Flexibility in Evaluation Order

- lazy values in a strict language → usually easy
- very easy with first-class functions
- including infinite data structures (co-data)

On the Other Hand

- strict values in a lazy language → usually hard
- typically needs language support
- often very far from intuitive
- compare normal forms: beta, beta-eta, head, weak head
- Haskell: `seq`, `deepseq`, `NFData`, `$!`, BangPatterns

Implementation: Recursive Evaluation

- the simplest method
- works directly on the AST
- may not need an explicit execution stack
- also the slowest

```
Value eval( If &e_if )
{
    if ( eval( e_if.condition ) )
        return eval( e_if._then );
    else
        return eval( e_if._else );
}
```

Reverse Polish Notation (RPN)

- faster than recursive
- only useful with eagerly evaluated constructs
- good for arithmetic-heavy programs
- recall postfix syntax from part 1
- `(5 + 3) * x` written as `5 3 + x *`
- trivial evaluation on an explicit stack

RPN: Implementation

```
void eval()
{
    if ( size() == 1 )
        return;
    Value a = pop(), b = pop();
    Op op = pop();
    if ( op == Add )
        push( a + b );
    // ...
}
```

- the result is the only value left on the stack

RPN: Control Flow

- control flow in an RPN evaluator is a bit tricky
- normally every “operator” is strict

However

- lazy semantics in a strict language? thunks
- push thunks for then/else branches onto the RPN stack
- profit

Function calls?

Three-Address Code

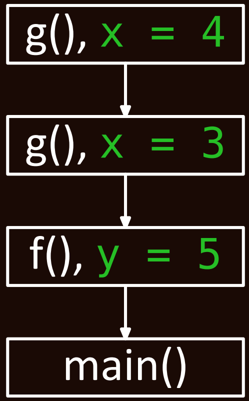
- might be faster than RPN
- control flow is straightforward
- ~halfway to a compiler
- data stored in arrays (not stacks)
- a lot more complicated than RPN
- quite some room for optimisation

Trampolines

- execute continuation-passing-style programs
- converts CPS into standard call/return semantics
- more of a compiler technique

Keeping Track of Calls

```
void g( int x )
{
    g( x + 1 );
}
void f() { int y; g( 3 ); }
int main() { f(); }
```



Implementation Strategies

- meta-circular (in a recursive evaluator)
- re-use the explicit RPN evaluation stack
- explicit “evaluation context” stack

Assignment (weeks 5 & 6)

- write an evaluator for your language
- arithmetic, conditionals, loops, variables and function calls
- mutable variables and an assignment operator
- write arithmetic- and recursion-based tests
- lexical closures are optional

Due 5th of April, 8am!

Assignment Hints

- a recursive evaluator is the simplest to implement
- strict evaluation order is the simplest
- you can keep variable values in an `std::unordered_map`
- RPN evaluation is also nice (don't forget about thunks)
- hybrids are viable (recursion only for calls & control flow)

Part 4: Type Checking

Overview

- what is a type
- sub-typing
- dynamic types (run-time checking)
- static types (ahead of time)
- classes and objects

Why Types?

- same reason as syntax checkers
- programmers (= people) make mistakes
- type mismatch is, usually, a mistake
- types = high-powered version of dimension analysis
- you don't want to add seconds to meters by mistake
- hence, type discipline and enforcement

What is a Type?

- first approximation: a set of values
- set of integers, set of strings, etc.
- every value belongs to a (single) type
- both values and variables have types

Function Types?

- eg. a set of maps from integers to integers
- maps are still sets, so this (almost) works out

Well-Typed Programs

- all type constraints are satisfied
- in particular, on function (operator) applications
- let $f :: T \to T$, $x :: T$ and $y :: U$
- `f x` is well-typed, `f y` is not
- also: assignment and initialisers, pattern matching

```
std::string x = 0.5; /* not well typed */
```

Products and Sums

- cartesian product of two types is again a type
- so is a sum (union, or maybe a disjoint union)
- unions + products form the basis of algebraic data types
- function type is a special subset of the product type

Multi-parameter functions

- $(T \times T) \to T$ is what C/C++ use
- $T \to (T \to T)$ is what Haskell uses
- the two are isomorphic (think curry/uncurry)

Product Types: Aggregates

- C `struct` is a typical product type
- a more “obvious” example: `std::pair` and `std::tuple`
- products with named fields are usually very important
- (also known as records)
- they form the backbone of user-defined types
- (classes are based on product types)

Subtyping

- maybe there's a user type `shape`
- every `circle` is clearly also a `shape`
- subtypes correspond to subsets
- induces a (pre)order relation on types

Contravariance

- let T be a type and S its subtype
- whenever a T is expected, S can be provided
- this usual behaviour is called covariant
- however! $T \rightarrow S$ is a subtype of $S \rightarrow S$
- function arguments are contravariant wrt. subtyping

Polymorphism

- monomorphic function types are quite constraining
- eg: plain C functions
- think `int min( int a, int b )` … how about `float`?
- counter-eg.: C++ function templates
- “types = sets” is no longer good enough

Approaches

- parametric: eg. Hindley-Milner
- ad-hoc: like parametric but dirtier (think C++ templates)
- subtyping + optionally late binding

Parametric Polymorphism

- one implementation, multiple (parametric) types
- ML, Haskell, etc. (based on Hindley-Milner)
- adds type variables
- `id :: a -> a` is good for any type `a`
- type checking is only a little harder than monomorphic
- C++ templates (w/o specialisation) are an approximation
- can be extended with type classes (Haskell)
- `min :: (Ord a) => a -> a -> a`

Algebraic Data Types (revisited)

- products and sums are nice but relatively weak
- how about recursive (infinite) data types?
- allows encoding lists, trees and other inductive types
- may also allow encoding co-data types
- `data List = Nil | Cons Int List`
- values must contain pointers

Parametric ADTs

- also: much more powerful with type variables
- `data List a = Nil | Cons a List`

Static Type Checking

- all type enforcement is done at compile/load time
- type information can be erased (more efficient execution)
- may require explicit type annotation (as in C, C++98)
- or be partially inferred (modern Haskell, C++11 and later)
- or be completely inferred ("classical" Haskell)
- type errors show up early
- may allow static (fast) type-based dispatch

Dynamic Type Checking

- type enforcement is (mostly) done at runtime
- values carry along their types encoded as data
- function application also runs the type checker
- RTTI could be as little as a couple of bits (LISP)
- or as much as a full machine pointer (OOP)

Classes and Objects

- subtyping naturally leads to OOP
- extends types with methods and encapsulation
- optionally with late binding
- one signature, multiple types, multiple implementations
- primarily a problem decomposition tool
- also neatly solves namespace problems
- works with static (C++) and dynamic (Python) type checkers

Late Binding

- supertype methods can be overridden in subtypes
- different implementations for different types
- form of run-time, type-based dispatch
- incompatible with (completely) static types
- in C++ realised through `vtable` pointers

Type Casting and Coercion

- sometimes you know you are right
- even though the types don't match
- casts convert from one type to another
- coercion simply re-interprets the value
- both more-or-less break type safety
- C has some arcane implicit casting rules

Assignment (weeks 7 & 8): Static Variant

- allow user-defined product types with named fields
- implement monomorphic function types
- add type annotations to the parser & AST
- type-check each function application at load time

Assignment: Dynamic/OOP Variant

- allow user-defined classes (with attributes and methods)
- pass values in the evaluator as references to objects
- implement late binding (type-based dispatch)
- detect failing method lookups at runtime

Due 19th of April, 8am! (optional)

Assignment Hints

- both variants need parser extensions
- dynamic types are easier to work with (from user POV)
- static types are safer and get you faster code
- static type checker builds on the semantic checker
- dynamic type checker builds on the evaluator
- you can mix & match aspects of both (like C++)
- it’s OK to put types and variables in a single namespace

Part 5: Memory Management

Overview

- what lives in memory
- reference counting
- mark and sweep
- copying collectors (compacting)
- Cheney on the M.T.A.
- generational collection
- latency and concurrency

What is in program's memory?

- scalar data and arrays of scalars
- data structures with pointers in them

Pointers: Good and Bad

- pointer dereferences are expensive
- allows encoding all sorts of structure
- lists, trees, graphs
- very useful for building abstractions

From Flat Memory to Objects

- imagine a node in a linked list
- it lives somewhere
- how do you decide where to put one?
- enter `malloc` and `free`

Semi-Automatic Memory Management

- `malloc` finds a good place to put data
- `free` marks a bit of memory for re-use

Building the Abstraction Tower

- `malloc`/`free` give us abstract-ish objects
- but we still need to track lifetime manually
- and worse, place `free` calls statically
- this is tedious and not always possible

Automatic Garbage Collection

- figure out which objects are alive (and which dead)
- we no longer need to call `free`
- `free` is dynamically performed by the GC

Basic Idea: Reachability along Pointers

- pick a root set of live objects
- could be the C stack + registers
- or the active (executing) frame
- live = reachable from the root set
- dead = everything else

First Approximation: Reference Counting

- along with each object, keep a counter
- when a pointer is created/copied, increase the counter
- when a pointer is lost, decrease the counter
- when the counter hits zero, `free` the object

Problems

- expensive to take/copy pointers (memory write)
- fails to free object cycles

Advantages

- low/predictable latency
- reasonable memory overhead

Garbage Collectors

- add a collector procedure
- the rest of the program is called the mutator
- run the collector at convenient times (not too often)

Dealing with Loops: Mark & Sweep

- the collector executes reachability along pointers
- marking every reachable object
- then iterating over all objects
- calling `free` on the unmarked ones (sweeping)

Challenges

- the collector procedure may need to allocate memory
- all mutator threads may need to stop while the collector runs
- the collector needs to know which words are pointers
- → problems with foreign function interfaces (C calls)
- performance under memory pressure

Mark & Sweep: Advantages

- comparatively easy to implement
- low memory overhead
- can re-use existing `malloc`/`free`
- approximate (conservative) collection is possible

Disadvantages

- high/messy latency (bad for interactive programs)
- more memory used = slower collection
- interacts badly with concurrency

A Copying Collector

- split memory into 2 halfspaces
- one is the working set, other is dormant
- bump allocation of new memory
- collect when the live halfspace fills up

Collection

- copy live objects to the other halfspace
- updating all pointers along the way

Cheney's Algorithm

- look at a from-space object
- copy it over to the to-space
- replace the from-space copy with a forwarding pointer
- recurse/update pointers in the to-space copy
- (not actually implemented recursively)

Copying Collectors: Advantages

- fast memory allocation
- no time spent dealing with garbage
- keeps data physically close together
- possibly improving cache utilisation

Disadvantages

- needs exact information about pointers
- poor memory utilisation (always 1 empty halfspace)

Cheney on the M.T.A.

- all allocation is done on the C stack
- when the stack is about to fill up:
- make a new stack and “Cheney” data from the old one
- the program is compiled into C
- the compiled functions never return
- easy integration with C calls

Disadvantage: same as “normal” copying collector

Compromises: Generational Collectors

- observation: many objects only live for a short while
- split memory into a hatchery and a mature space
- use a different collector for each
- typical: mark & copy for the hatchery (minor collection)
- mark & sweep for the mature space (major collection)
- the hatchery is traced much more often

Generational Collectors: Advantages

- short-lived objects are quickly eliminated
- hot data is kept together (good for CPU caches)
- minor collection is fast & predictable (wrt. latency)
- foreign objects can live in the (non-moving) mature space

Disadvantages

- more complicated
- does not fix all the problems

A Note on Latency: Incremental Collection

- latency in interactive applications is bad
- even more so in real-time systems
- also in distributed computations
- interleave the mutator and the collector
- incremental collector can be made real-time
- (by imposing a deadline on the increment)
- tricky, but easier than concurrent collection

Concurrent Collectors

- concurrent data structures are hard
- not freeing dead objects is not a big problem
- (they will be picked up by a later cycle)
- freeing live objects is a big problem
- needs cooperation from mutator threads
- easy-ish with reference counting

Eg. `http://www.aicas.com/papers/ismm02f-siebert.pdf`

Assignment (weeks 9 & 10):

- implement a garbage collector
- (optional, deadline May the 3rd, 8am)

Part 6: Talking to The Outside World

Overview

- foreign function interface (FFI)
- constructing calls
- dealing with memory & outputs
- aggregate arguments and return values
- a simple runtime-only solution

Foreign Function Interface

- a mechanism for calling procedures
- defined in a language different from our own
- a typical target language is C
- crucial for re-use of existing code

Constructing Calls

- problem: we need to call a function
- but we don't know what its arguments are

Some options:

- ad-hoc: hard-code some argument combinations
- template metaprogramming
- automatic code generation
- re-implement the C calling convention

An Ad-Hoc Approach

- good enough to cover most syscalls
- not good to talk to libraries

```
void ccall( void (*f)(), ArgT argt, ArgV v )
{
    if ( argt == ArgT{ Int } )
        return f( v[ 0 ].asInt() );
    if ( argt == ArgT{ Int, Int } )
        return f( v[ 0 ].asInt(), v[ 1 ].asInt() );
    /* ... */
}
```

Template Metaprogramming

- use variadic/recursive function templates
- automates data conversion (the `asInt()` bit)
- required instances need to be known at compile time
- does not really fix the problem

```
int f( int a, int b, const char *c );
auto tup = std::make_tuple( 3, 7, "foo" );
brick::tuple::pass( f, tup ); // call f( 3, 7, "foo" )
```

Automatic Code Generation

- uses specific, per-function wrappers
- the wrappers are generated as C (C++) code
- may use the template approach to simplify generated code

```
Value wrap_write( std::vector< Value > args )
{
    int rv = write( args[ 0 ].asInt(),
                    args[ 1 ].asString(),
                    args[ 2 ].asInt() );
    return Value( rv );
}
```

The C Calling Convention

- architecture-specific (x86 is simple, amd64 is complex)
- the generic `ccall` needs to be written in assembly
- most compact but least portable

x86/cdecl

- arguments go onto the stack (right-to-left)
- scalar return values either in `eax` or `st0`

amd64: a 10 page spec on what goes where

Dealing with Outputs

- some functions return variable-sized data
- like the `read` function
- using output arguments (represented by pointers)
- such arguments must be treated differently
- the output of `read` should give us an in-language string

```
char buffer[32];
read( 0, buffer, 32 );
// buffer contains the data we want
```

Aggregate Values

- C supports passing structures as arguments
- and returning them as values
- will not work with the ad-hoc approach
- too many different sizes

```
struct foo { int x, y, z; double bar; };
foo update_foo( foo x ) { ... }
```

A Simple Approach

- usable with ad-hoc call construction
- does not need to invoke a compiler
- looks up functions by using `dlsym`

```
int main()
{
    int (*w)() = dlsym( NULL, "write" );
    w( 1, "hello world\n", 12 );
}
```

How to Construct Wrappers

- option 1: reconstruct from calls
- will not work for variadic functions (`printf`)
- option 2: special syntax for declaring C functions
- you rely on the user to translate the prototypes
- option 3: parse C headers (hard)

```
(define fun
 (foreign-lambda c-string "fun" c-string int))
```

Assignment

- implement a simple C-based FFI for your interpreter
- must be able to call integer-argument functions w/ up to 4 args
- it must be able to deal with `read` or similar
- use the FFI to get I/O capabilities
- implement an interactive game of tic-tac-toe

There is no deadline.