Why Study Compilers?

• Although many of the basic ideas were developed over 50 years ago, compiler construction is still an evolving and active area of research and development.

• Compilers are intimately related to programming language design and evolution.

• Compilers are a Computer Science success story illustrating the hallmarks of our field --- higher-level abstractions implemented with lower-level abstractions.

• Every Computer Scientist should have a basic understanding of how compilers work.
Compilation is a special kind of translation

Source Program Text

The compiler

program for target “machine”

Just text – no way to run program!

We have a “machine” to run this!

A good compiler should...

- be correct in the sense that meaning is preserved
- produce usable error messages
- generate efficient code
- itself be efficient
- be well-structured and maintainable

This course! OptComp, Part II

Pick any 2? Just 1?
Mind The Gap

<table>
<thead>
<tr>
<th>High Level Language</th>
<th>Typical Target Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Machine” independent</td>
<td>“Machine” specific</td>
</tr>
<tr>
<td>Complex syntax</td>
<td>Simple syntax</td>
</tr>
<tr>
<td>Complex type system</td>
<td>Simple types</td>
</tr>
<tr>
<td>Variables</td>
<td>memory, registers, words</td>
</tr>
<tr>
<td>Nested scope</td>
<td>Single flat scope</td>
</tr>
<tr>
<td>Procedures, functions</td>
<td></td>
</tr>
<tr>
<td>Objects</td>
<td></td>
</tr>
<tr>
<td>Modules</td>
<td></td>
</tr>
<tr>
<td>…</td>
<td></td>
</tr>
</tbody>
</table>

Help!!! Where do we begin???
The Gap, illustrated

```java
public class Fibonacci {
    public static long fib(int m) {
        if (m == 0) return 1;
        else if (m == 1) return 1;
        else return fib(m - 1) + fib(m - 2);
    }

    public static void main(String[] args) {
        int m = Integer.parseInt(args[0]);
        System.out.println(fib(m) + "\n");
    }
}
```

```java
public class Fibonacci {
    public Fibonacci();
    Code:
        0: aload_0
        1: invokespecial #1
        4: return
    public static long fib(int);
    Code:
        0: iload_0
        1: ifne            6
        4: lconst_1
        5: lreturn
        6: iload_0
        7: iconst_1
        8: if_icmpne      13
        11: lconst_1
        12: lreturn
        13: iload_0
        14: iconst_1
        15: isub
        16: invokestatic #2
        19: iload_0
        20: iconst_2
        21: isub
        22: invokestatic #2
        25: ladd
        26: lreturn
    public static void main(java.lang.String[]);
    Code:
        0: aload_0
        1: iconst_0
        2: aaload
        3: invokestatic #3
        6: istore_1
        7: getstatic    #4
        10: new           #5
        13: dup
        14: invokespecial #6
        17: iload_1
        18: invokestatic #2
        21: invokevirtual #7
        24: ldc           #8
        26: invokevirtual #9
        29: invokevirtual #10
        32: invokevirtual #11
        35: return
```

```
javac Fibonacci.java
javap -c Fibonacci.class
```
fib.ml

(* fib : int -> int *)
let rec fib m =
  if m = 0
  then 1
  else if m = 1
       then 1
  else fib(m - 1) + fib(m - 2)

ocamlc -dinstr fib.ml

OCaml VM bytecodes
The Gap, illustrated

fib.c

#include<stdio.h>

int Fibonacci(int);
int main()
{
    int n;
    scanf("%d",&n);
    printf("%d\n", Fibonacci(n));
    return 0;
}

int Fibonacci(int n)
{
    if ( n == 0 ) return 0;
    else if ( n == 1 ) return 1;
    else return ( Fibonacci(n-1) + Fibonacci(n-2) );
}

gcc -S fib.c
.section .TEXT, __text, regular, pure_instructions
.globl _main
.align 4, 0x90
_main:  ## @main
    .cfi_startproc
    ## BB#0:
    pushq  %rbp
    Ltmp2:
        .cfi_def_cfa_offset 16
Lttmp3:  .cfi_offset %rbp, -16
    movq  %rsp, %rbp
    Ltmp4:
        .cfi_def_cfa_register %rbp
    subq  $16, %rsp
    leaq  L_.str(%rip), %rdi
    leaq  -8(%rbp), %rsi
    movl  %edx, -4(%rbp)
    movb  %dl, %al
    callq _scanf
    movl  -8(%rbp), %edi
    movl  %eax, -12(%rbp)         ## 4-byte Spill
    callq _Fibonacci
    movl  %eax, %edi
    callq _Fibonacci
    movl  %edi, -4(%rbp)
    LBB1_5:
        movl  -4(%rbp), %eax
        addq  $16, %rsp
        popq  %rbp
        ret
    .cfi_endproc

.globl _Fibonacci
.align 4, 0x90
_Fibonacci:  ## @Fibonacci
    .cfi_startproc
    ## BB#0:
    pushq  %rbp
    Ltmp7:  .cfi_def_cfa_offset 16
Lttmp8:  .cfi_offset %rbp, -16
    movq  %rsp, %rbp
    Ltmp9:

L_.str:  ## @.str
        .asciz "%d"
L_.str1:  ## @.str1
        .asciz "%d\n"

.section .TEXT, __cstring, cstring_literals
_x86/Mac OS
Conceptual view of a typical compiler

The compiler

Source Program Text

Front End

Middle

Back End

ISA = Instruction Set Architecture

ISA/OS targeted code (x86/unix, ...)

errors, warnings

Key to bridging The Gap: divide and conquer. The Big Leap is broken into small steps. Each step broken into yet smaller steps ...

ISA/OS independent “byte code”

Virtual Machine (VM) examples: JVM, Dalvik, .NET CLR

Operating System (OS)
The AST output from the front-end should represent a legal program in the source language. (“Legal” of course does not mean “bug-free”!)
Trade-off: with more optimisations the generated code is (normally) **faster**, but the compiler is **slower**
Our view of the middle
a sequence of small transformations

Intermediate Languages

IL-1 \rightarrow IL-2 \rightarrow \cdots \rightarrow IL-k

- Each IL has its own semantics (perhaps informal)
- Each transformation (\rightarrow) preserves semantics (SPL!)
- Each transformation eliminates only a few aspects of the gap
- Each transformation is fairly easy to understand
- Some transformations can be described as “optimizations”
- We will associate each IL with its own interpreter/VM. (Again, not something typically done in “industrial-strength” compilers.)

Of course industrial-strength compilers may collapse many small-steps ...
The back-end

Low-level retargetable representation → Back-end

- JVM bytecodes
- x86/Linux
- x86/MacOS
- x86/FreeBSD
- x86/Windows
- ARM/Android
- ...
- ...

- Requires intimate knowledge of instruction set and details of target machine
- When generating assembler, need to understand details of OS interface
- Target-dependent optimisations happen here!
Compilers must be compiled

Something to ponder:
A compiler is just a program. But how did it get compiled? The OCaml compiler is written in OCaml.

How was the compiler compiled?
Approach Taken

- We will develop a compiler for a fragment of L3 introduced in Semantics of Programming Languages, Part 1B.
- We will pay special attention to the correctness.
- We will compile only to Virtual Machines (VMs) of various kinds. See Part II optimising compilers for generating lower-level code.
- Our toy compiler is available on the course web site.
- We will be using the OCaml dialect of ML.

- Install from https://ocaml.org.
- See OCaml Labs: http://www.cl.cam.ac.uk/projects/ocamllabs.
SML Syntax vs. OCaml Syntax

datatype 'a tree =
    Leaf of 'a
  | Node of 'a * ('a tree) * ('a tree)

fun map_tree f (Leaf a) = Leaf (f a)
  | map_tree f (Node (a, left, right)) =
    Node(f a, map_tree f left, map_tree f right)

let val l =
  map_tree (fn a => [a]) [Leaf 17, Leaf 21]
in
  List.rev l
end

type 'a tree =
    Leaf of 'a
  | Node of 'a * ('a tree) * ('a tree)

let rec map_tree f = function
  | Leaf a -> Leaf (f a)
  | Node (a, left, right) ->
    Node(f a, map_tree f left, map_tree f right)

let l =
  map_tree (fun a -> [a]) [Leaf 17; Leaf 21]
in
  List.rev l
1. Overview
3. Lexical analysis: application of Theory of Regular Languages and Finite Automata
4. Generating Recursive descent parsers
5. Beyond Recursive Descent Parsing I
6. Beyond Recursive Descent Parsing II
7. High-level “definitional” interpreter (interpreter 0). Make the stack explicit and derive interpreter 2
8. Flatten code into linear array, derive interpreter 3
9. Move complex data from stack into the heap, derive the Jargon Virtual Machine (interpreter 4)
11. A few program transformations. Tail Recursion Elimination (TRE), Continuation Passing Style (CPS). Defunctionalisation (DFC)
12. CPS+TRE+DFC provides a formal way of understanding how we went from interpreter 0 to interpreter 2. We fill the gap with interpreter 1
13. Assorted topics: compilation units, linking. From Jargon to x86
14. Assorted topics: simple optimisations, OOP object representation
15. Run-time environments, automated memory management (“garbage collection”)
16. Bootstrapping a compiler
• Slang (= Simple LANGuage)
  – A subset of L3 from Semantics …
  – … with very ugly concrete syntax
  – You are invited to experiment with improvements to this concrete syntax.

• Slang : concrete syntax, types

• Abstract Syntax Trees (ASTs)

• The Front End

• A short in-lecture demo of slang and a brief tour of the code …
Clunky Slang Syntax (informal)

\[ \begin{align*}
uop & ::= - \mid \sim \\
bop & ::= + \mid - \mid * \mid < \mid = \mid \&\& \mid ||
\end{align*} \]

\[ \begin{align*}
t & ::= \text{bool} \mid \text{int} \mid \text{unit} \mid (t) \mid t * t \mid t + t \mid t \rightarrow t \mid t \text{ ref}
\end{align*} \]

\[ \begin{align*}
e & ::= () \mid n \mid \text{true} \mid \text{false} \mid x \mid (e) \mid ? \mid e \ bop \ e \mid uop \ e \\
& \quad \mid \text{if } e \ \text{then } \text{else } e \ \text{end} \\
& \quad \mid e \ e \mid \text{fun } (x : t) \rightarrow e \ \text{end} \\
& \quad \mid \text{let } x : t = e \ \text{in } e \ \text{end} \\
& \quad \mid \text{let } f(x : t) : t = e \ \text{in } e \ \text{end} \\
& \quad \mid !e \mid \text{ref } e \mid e := e \mid \text{while } e \ \text{do } e \ \text{end} \\
& \quad \mid \text{begin } e; e; \ldots \ e \ \text{end} \\
& \quad \mid (e, e) \mid \text{snd } e \mid \text{fst } e \\
& \quad \mid \text{inl } t \ e \mid \text{inr } t \ e \\
& \quad \mid \text{case } e \ \text{of } \text{inl}(x : t) \rightarrow e \mid \text{inr}(x : t) \rightarrow e \ \text{end}
\end{align*} \]
From slang/examples

let fib( m : int) : int = 
  if m = 0 
  then 1 
  else if m = 1 
    then 1 
    else fib (m - 1) + 
      fib (m -2) 
  end 
end 
in 
  fib(?) 
end

let gcd( p : int * int) : int = 
  let m : int = fst p 
  in let n : int = snd p 
  in if m = n 
    then m 
    else if m < n 
      then gcd(m, n - m) 
      else  gcd(m - n, n) 
  end 
end 
in gcd(?, ?) end

The ? requests an integer input from the terminal
Input file foo.slang

Parse (we use Ocaml versions of LEX and YACC, covered in Lectures 3 --- 6)

Parsed AST (Past.expr)

Static analysis: check types, and context-sensitive rules, resolve overloaded operators

Parsed AST (Past.expr)

Remove “syntactic sugar”, file location information, and most type information

Intermediate AST (Ast.expr)
type var = string

type loc = Lexing.position

type type_expr =
  | TEint
  | TBo bool
  | TEunit
  | TRef of type_expr
  | TArrow of type_expr * type_expr
  | TProduct of type_expr * type_expr
  | TUnion of type_expr * type_expr

type oper = ADD | MUL | SUB | LT |
  | AND | OR | EQ | EQB | EQI

type unary_oper = NEG | NOT

type expr =
  | Unit of loc
  | What of loc
  | Var of loc * var
  | Integer of loc * int
  | Boolean of loc * bool
  | UnaryOp of loc * unary_oper * expr
  | Op of loc * expr * oper * expr
  | If of loc * expr * expr * expr
  | Pair of loc * expr * expr
  | Fst of loc * expr
  | Snd of loc * expr
  | Inl of loc * type_expr * expr
  | Inr of loc * type_expr * expr
  | Case of loc * expr * lambda * lambda
  | While of loc * expr * expr
  | Seq of loc * (expr list)
  | Ref of loc * expr
  | Deref of loc * expr
  | Assign of loc * expr * expr
  | Lambda of loc * lambda
  | App of loc * expr * expr
  | Let of loc * var * type_expr * expr * expr
  | LetFun of loc * var * lambda
  | LetRecFun of loc * var * lambda
  | type_expr * expr

Locations (loc) are used in generating error messages.
val infer : (Past.var * Past.type_expr) list -> (Past.expr * Past.type_expr)

val check : Past.expr -> Past.expr (* infer on empty environment *)

• Check type correctness
• Rewrite expressions to resolve EQ to EQI (for integers) or EQB (for bools).
• Only LetFun is returned by parser. Rewrite to LetRecFun when function is actually recursive.

Lesson : while enforcing “context-sensitive rules” we can resolve ambiguities that cannot be specified in context-free grammars.
type var = string

type oper = ADD | MUL | SUB | LT | AND | OR | EQB | EQI

type unary_oper = NEG | NOT | READ

No locations, types. No Let, EQ.

Is getting rid of types a bad idea? Perhaps a full answer would be language-dependent...

<table>
<thead>
<tr>
<th>expr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit</td>
</tr>
<tr>
<td>Var of var</td>
</tr>
<tr>
<td>Integer of int</td>
</tr>
<tr>
<td>Boolean of bool</td>
</tr>
<tr>
<td>UnaryOp of unary_oper * expr</td>
</tr>
<tr>
<td>Op of expr * oper * expr</td>
</tr>
<tr>
<td>If of expr * expr * expr</td>
</tr>
<tr>
<td>Pair of expr * expr</td>
</tr>
<tr>
<td>Fst of expr</td>
</tr>
<tr>
<td>Snd of expr</td>
</tr>
<tr>
<td>Inl of expr</td>
</tr>
<tr>
<td>Inr of expr</td>
</tr>
<tr>
<td>Case of expr * lambda * lambda</td>
</tr>
<tr>
<td>While of expr * expr</td>
</tr>
<tr>
<td>Seq of (expr list)</td>
</tr>
<tr>
<td>Ref of expr</td>
</tr>
<tr>
<td>Deref of expr</td>
</tr>
<tr>
<td>Assign of expr * expr</td>
</tr>
<tr>
<td>Lambda of lambda</td>
</tr>
<tr>
<td>App of expr * expr</td>
</tr>
<tr>
<td>LetFun of var * lambda * expr</td>
</tr>
<tr>
<td>LetRecFun of var * lambda * expr</td>
</tr>
</tbody>
</table>

and lambda = var * expr
This is done to simplify some of our code. Is it a good idea? Perhaps not.
1. Theory of Regular Languages and Finite Automata applied to lexical analysis.
2. Context-free grammars
3. The ambiguity problem
4. Generating Recursive descent parsers
5. Beyond Recursive Descent Parsing I
6. Beyond Recursive Descent Parsing II
What problem are we solving?

Translate a sequence of characters

if m = 0 then 1 else if m = 1 then 1 else fib (m - 1) + fib (m - 2)

into a sequence of tokens

IF, IDENT “m”, EQUAL, INT 0, THEN, INT 1, ELSE, IF, IDENT “m”, EQUAL, INT 1, THEN, INT 1, ELSE, IDENT “fib”, LPAREN, IDENT “m”, SUB, INT 1, RPAREN, ADD, IDENT “fib”, LPAREN, IDENT “m”, SUB, INT 2, RPAREN

implemented with some data type

type token =
| INT of int| IDENT of string | LPAREN | RPAREN
| ADD | SUB | EQUAL | IF | THEN | ELSE
| ...
Regular expressions (concrete syntax)

over a given alphabet $\Sigma$.

Let $\Sigma'$ be the 4-element set $\{\epsilon, \emptyset, |, \ast\}$ (assumed disjoint from $\Sigma$)

$$U = (\Sigma \cup \Sigma')^*$$

axioms: _______ _______ _______

$\epsilon$ $\emptyset$

rules: _______ _______ _______

$(r)$ $rs$ $r^*$

(where $a \in \Sigma$ and $r, s \in U$)
Recall from Discrete Mathematics (Part 1A)

Example of a finite automaton

\[ M \triangleq \]

- set of states: \( \{q_0, q_1, q_2, q_3\} \)
- input alphabet: \( \{a, b\} \)
- transitions, labelled by input symbols: as indicated by the above directed graph
- start state: \( q_0 \)
- accepting state(s): \( q_3 \)
Recall from Discrete Mathematics (Part 1A)

Kleene’s Theorem

**Definition.** A language is **regular** iff it is equal to $L(M)$, the set of strings accepted by some deterministic finite automaton $M$.

**Theorem.**

(a) For any regular expression $r$, the set $L(r)$ of strings matching $r$ is a regular language.

(b) Conversely, every regular language is the form $L(r)$ for some regular expression $r$. 
Given a regular expression, 

\[ e \]

and an input string \( w \), determine if \( w \in L(e) \).

Construct a DFA \( M \) from \( e \) and test if it accepts \( w \).

Recall construction: regular expression \( \rightarrow \) NFA \( \rightarrow \) DFA
Given an ordered list of regular expressions,

\[ e_1, e_2, \ldots, e_k \]

and an input string \( w \), find a list of pairs

\[ (i_1, w_1), (i_2, w_2), \ldots, (i_n, w_n) \]

such that

1) \( w = w_1 w_2 \ldots w_n \)

2) \( w_j \in L(e_{i_j}) \)

3) \( w_j \in L(e_s) \Rightarrow i_j \leq s \) (priority rule)

4) \( \forall j : \forall u \in \text{prefix}(w_{j+1} w_{j+2} \ldots w_n) : u \neq \varepsilon \)

\[ \Rightarrow \forall s : w_j u \notin L(e_s) \) (longest match) \]
Define Tokens with Regular Expressions (Finite Automata)

Keyword: if

This FA is really shorthand for:

"dead state"
<table>
<thead>
<tr>
<th>Regular Expression</th>
<th>Finite Automata</th>
<th>Token</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Keyword:</strong> if</td>
<td><img src="#" alt="Automata Diagram" /></td>
<td>KEY(IF)</td>
</tr>
<tr>
<td><strong>Keyword:</strong> then</td>
<td><img src="#" alt="Automata Diagram" /></td>
<td>KEY(then)</td>
</tr>
<tr>
<td><strong>Identifier:</strong> [a-zA-Z][a-zA-Z0-9]⁺</td>
<td><img src="#" alt="Automata Diagram" /></td>
<td>ID(s)</td>
</tr>
</tbody>
</table>
Define Tokens with Regular Expressions (Finite Automata)

<table>
<thead>
<tr>
<th>Regular Expression</th>
<th>Finite Automata</th>
<th>Token</th>
</tr>
</thead>
<tbody>
<tr>
<td>number: [0-9][0-9]*</td>
<td><img src="image1" alt="Automata Diagram" /></td>
<td>NUM(n)</td>
</tr>
<tr>
<td>real: (([0-9]+\ \cdot\ ([0-9]+)))</td>
<td><img src="image2" alt="Automata Diagram" /></td>
<td>NUM(n)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
No Tokens for “White-Space”

White-space:
(‘ ‘ | ‘\n’ | ‘\t’) +
| ‘%’ [A-Za-z0-9‘] + ‘\n’
Constructing a Lexer

INPUT: an ordered list of regular expressions

\[ e_1 \]
\[ e_2 \]
\[ \vdots \]
\[ e_k \]

Construct all corresponding finite automata

\[ NFA_1 \]
\[ NFA_2 \]
\[ \vdots \]
\[ NFA_k \]

Construct a single non-deterministic finite automata

Construct a single deterministic finite automata

\[ \text{NFA} \]
\[ \text{DFA} \]

(1) Keyword : then
(2) Ident : \([a-z][a-z]^*\)
(2) White-space: ‘ ‘
What about longest match?

Start in initial state,
Repeat:
  (1) read input until dead state is reached. Emit token associated with last accepting state.
  (2) reset state to start state.

| = current position, $ = EOF

Input

|then thenx$ 1  0
|then thenx$ 2  2
|then en thenx$ 3  3
|then n thenx$ 4  4
|then| thenx$ 5  5
|then |thenx$ 0  5 EMIT KEY(THEN)
|then| thenx$ 1  0 RESET
|then| thenx$ 7  7
|then t|henx$ 0  7 EMIT WHITE(‘ ‘)
|then| thenx$ 1  0 RESET
|then t|henx$ 2  2
|then th|enx$ 3  3
|then the|nx$ 4  4
|then then|x$ 5  5
|then thenx|$ 6  6
|then thenx$| 0  6 EMIT ID(thenx)
Concrete vs. Abstract Syntax Trees

Parse tree = derivation tree = concrete syntax tree

Abstract Syntax Tree (AST)

An AST contains only the information needed to generate an intermediate representation

Normally a compiler constructs the concrete syntax tree only implicitly (in the parsing process) and explicitly constructs an AST.
On to Context Free Grammars (CFGs)

E ::= ID
E ::= NUM
E ::= E * E
E ::= E / E
E ::= E + E
E ::= E – E
E ::= ( E )

E is a *non-terminal symbol*

ID and NUM are *lexical classes*

*, (, ), +, and – are *terminal symbols*.

E ::= E + E is called a *production rule*.

Usually will write this way:

E ::= ID | NUM | E * E | E / E | E + E | E – E | ( E )
CFG Derivations

(G1) \[ E ::= \text{ID} \mid \text{NUM} \mid \text{ID} \mid E \cdot E \mid E / E \mid E + E \mid E - E \mid (E) \]

\[ E \rightarrow E \cdot E \]
\[ E \rightarrow E \cdot (E) \]
\[ E \rightarrow E \cdot (E - E) \]
\[ E \rightarrow E \cdot (E - 10) \]
\[ E \rightarrow E \cdot (2 - 10) \]
\[ E \rightarrow (E) \cdot (2 - 10) \]
\[ E \rightarrow (E + E) \cdot (2 - 10) \]
\[ E \rightarrow (E + 4) \cdot (2 - E) \]
\[ E \rightarrow (17 + 4) \cdot (2 - 10) \]

The Derivation Tree for \((17 + 4) \cdot (2 - 10)\)
A CFG is a quadruple $G = (N, T, R, S)$ where
- $N$ is the set of non-terminal symbols
- $T$ is the set of terminal symbols ($N$ and $T$ disjoint)
- $S \in N$ is the start symbol
- $R \subseteq N \times (N \cup T)^*$ is a set of rules

Example: The grammar of nested parentheses $G = (N, T, R, S)$ where
- $N = \{S\}$
- $T = \{(, )\}$
- $R = \{(S, (S)), (S, SS), (S, )\}$

We will normally write $R$ as $S ::= (S) \mid SS \mid$
Derivations, more formally...

• Start from start symbol (S)
• Productions are used to derive a sequence of tokens from the start symbol
• For arbitrary strings $\alpha$, $\beta$ and $\gamma$ comprised of both terminal and non-terminal symbols, and a production $A \rightarrow \beta$, a single step of derivation is $\alpha A \gamma \Rightarrow \alpha \beta \gamma$
  – i.e., substitute $\beta$ for an occurrence of $A$
\forall \alpha \Rightarrow^* \beta$ means that $b$ can be derived from $a$ in $0$ or more single steps
\forall \alpha \Rightarrow^+ \beta$ means that $b$ can be derived from $a$ in $1$ or more single steps
The language generated by $G$ is the set of all terminal strings derivable from the start symbol $S$:

$$L(G) = \left\{ w \in T^* \mid S \Rightarrow +w \right\}$$

For any subset $W$ of $T^*$, if there exists a CFG $G$ such that $L(G) = W$, then $W$ is called a Context-Free Language (CFL) over $T$. 
Ambiguity

Both derivation trees correspond to the string

$$1 + 2 \times 3$$

This type of ambiguity will cause problems when we try to go from strings to derivation trees!
Problem: Generation vs. Parsing

- Context-Free Grammars (CFGs) describe how to to **generate**
- **Parsing** is the inverse of generation,
  - Given an input string, is it in the language generated by a CFG?
  - If so, construct a derivation tree (normally called a **parse tree**).
  - Ambiguity is a big problem

Note: recent work on Parsing Expression Grammars (PEGs) represents an attempt to develop a formalism that describes parsing directly. This is beyond the scope of these lectures …
We can often modify the grammar in order to eliminate ambiguity

\[
(S) \\
S ::= E$
\]

\[
E ::= E + T \\
\quad | \ E - T \\
\quad | \ T
\]

\[
T ::= T * F \\
\quad | \ T / F \\
\quad | \ F
\]

\[
F ::= \text{NUM} \\
\quad | \text{ID} \\
\quad | \ ( \ E )
\]

This is the unique derivation tree for the string

\[1 + 2 * 3$

Note: \(L(G1) = L(G2)\). Can you prove it?
(G3) \( S ::= \text{if } E \text{ then } S \text{ else } S \mid \text{if } E \text{ then } S \mid \text{blah-blah} \)

What does

\[ \text{if } e_1 \text{ then if } e_2 \text{ then } s_1 \text{ else } s_3 \]

mean?

\begin{center}
\begin{tikzpicture}
  \node (S) {S};
  \node (E) [below left of=S] {E};
  \node (then) [below right of=E] {then};
  \node (S1) [below of=then] {S};
  \node (S2) [above of=then] {S};
  \node (else) [right of=S1] {else};
  \node (S3) [below of=else] {S};
  \draw (S) -- (E);
  \draw (S) -- (then);
  \draw (then) -- (S1);
  \draw (then) -- (S2);
  \draw (else) -- (S3);
  \end{tikzpicture}
\end{center}

OR

\begin{center}
\begin{tikzpicture}
  \node (S) {S};
  \node (E) [below left of=S] {E};
  \node (then) [below right of=E] {then};
  \node (S1) [below of=then] {S};
  \node (S2) [above of=then] {S};
  \node (else) [right of=S1] {else};
  \node (S3) [below of=else] {S};
  \node (S4) [right of=S3] {S};
  \node (S5) [right of=S4] {S};
  \node (S6) [right of=S5] {S};
  \draw (S) -- (E);
  \draw (S) -- (then);
  \draw (then) -- (S1);
  \draw (then) -- (S2);
  \draw (else) -- (S3);
  \draw (else) -- (S4);
  \draw (S4) -- (S5);
  \draw (S5) -- (S6);
  \end{tikzpicture}
\end{center}
(G4)
S ::= WE | NE
WE ::= if E then WE else WE | blah-blah
NE ::= if E then S
    | if E then WE else NE

Now,

if e1 then if e2 then s1 else s3

has a unique derivation.

Note: L(G3) = L(G4). Can you prove it?
(1) Some context free languages are inherently ambiguous --- every context-free grammar for it will be ambiguous. For example:

\[ L = \{ a^n b^n c^m d^m \mid m \geq 1, n \geq 1 \} \cup \{ a^n b^m c^m d^n \mid m \geq 1, n \geq 1 \} \]

(2) Checking for ambiguity in an arbitrary context-free grammar is not decidable! Ouch!

(3) Given two grammars G1 and G2, checking L(G1) = L(G2) is not decidable! Ouch!
The idea: use regular expressions as the basis of a lexical specification. The core of the lexical analyzer is then a deterministic finite automata (DFA)
Predictive (Recursive Descent) Parsing
Can we automate this?

```plaintext
int tok = getToken();

void advance() {tok = getToken();}
void eat (int t) {if (tok == t) advance(); else error();}
void S() {switch(tok) {
    case IF:    eat(IF); E(); eat(THEN);
                S(); eat(ELSE); S(); break;
    case BEGIN: eat(BEGIN); S(); L(); break;
    case PRINT: eat(PRINT); E(); break;
    default: error();
}}

void L() {switch(tok) {
    case END:  eat(END); break;
    case SEMI: eat(SEMI); S(); L(); break;
    default: error();
}}

void E() {eat(NUM) ; eat(EQ); eat(NUM); }
```

Parse corresponds to a left-most derivation constructed in a “top-down” manner

From Andrew Appel, “Modern Compiler Implementation in Java” page 46
Immediate left-recursion

\[ A ::= A\alpha_1 | A\alpha_2 | \ldots | A\alpha_k | \beta_1 | \beta_2 | \ldots | \beta_n \]

\[ A ::= \beta_1 A' | \beta_2 A' | \ldots | \beta_n A' \]

\[ A' ::= \alpha_1 A' | \alpha_2 A' | \ldots | \alpha_k A' | \varepsilon \]

For eliminating left-recursion in general, see Aho and Ullman.
Eliminating Left Recursion

Note that
\[ E ::= T \text{ and } E ::= E + T \]
will cause problems
since FIRST(T) will be included
in FIRST(E + T) ---- so how can
we decide which production
To use based on next token?

Solution: eliminate “left recursion”!
\[ E ::= T E' \]
\[ E' ::= + T E' \]
\[ T ::= F T' \]
\[ T' ::= * F T' \]
\[ F ::= \text{NUM} \]
\[ (E) \]

(G6)
\[ S ::= E$ \]
\[ E ::= T E' \]
\[ E' ::= + T E' | - T E' | \]
\[ T ::= F T' \]
\[ T' ::= * F T' | / F T' | \]
\[ F ::= \text{NUM} | \text{ID} | (E) \]
FIRST and FOLLOW

For each non-terminal X we need to compute

FIRST[X] = the set of terminal symbols that can begin strings derived from X

FOLLOW[X] = the set of terminal symbols that can immediately follow X in some derivation

nullable[X] = true of X can derive the empty string, false otherwise

nullable[Z] = false, for Z in T

nullable[Y1 Y2 ... Yk] = nullable[Y1] and ... nullable[Yk], for Y(i) in N union T.

FIRST[Z] = {Z}, for Z in T

FIRST[ X Y1 Y2 ... Yk] = FIRST[X] if not nullable[X]

FIRST[ X Y1 Y2 ... Yk] = FIRST[X] union FIRST[Y1 ... Yk] otherwise
Computing First, Follow, and nullable

For each terminal symbol Z
  \( \text{FIRST}[Z] := \{Z\}; \)
  \( \text{nullable}[Z] := \text{false}; \)

For each non-terminal symbol X
  \( \text{FIRST}[X] := \text{FOLLOW}[X] := \{\}; \)
  \( \text{nullable}[X] := \text{false}; \)

repeat
  for each production \( X \rightarrow Y_1 Y_2 \ldots Y_k \)
    if \( Y_1, \ldots Y_k \) are all nullable, or \( k = 0 \)
      then \( \text{nullable}[X] := \text{true} \)
    for each \( i \) from 1 to \( k \), each \( j \) from \( i + 1 \) to \( k \)
      if \( Y_1 \ldots Y(i-1) \) are all nullable or \( i = 1 \)
        then \( \text{FIRST}[X] := \text{FIRST}[X] \cup \text{FIRST}[Y(i)] \)
      if \( Y(i+1) \ldots Y_k \) are all nullable or if \( i = k \)
        then \( \text{FOLLOW}[Y(i)] := \text{FOLLOW}[Y(i)] \cup \text{FOLLOW}[X] \)
    if \( Y(i+1) \ldots Y(j-1) \) are all nullable or \( i+1 = j \)
      then \( \text{FOLLOW}[Y(i)] := \text{FOLLOW}[Y(i)] \cup \text{FIRST}[Y(j)] \)
  until there is no change
### First, Follow, nullable table for G6

<table>
<thead>
<tr>
<th>Nullable</th>
<th>FIRST</th>
<th>FOLLOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>False</td>
<td>{ (, ID, NUM }</td>
</tr>
<tr>
<td>E</td>
<td>False</td>
<td>{ (, ID, NUM }</td>
</tr>
<tr>
<td>E'</td>
<td>True</td>
<td>{ +, - }</td>
</tr>
<tr>
<td>T</td>
<td>False</td>
<td>{ (, ID, NUM }</td>
</tr>
<tr>
<td>T'</td>
<td>True</td>
<td>{ *, / }</td>
</tr>
<tr>
<td>F</td>
<td>False</td>
<td>{ (, ID, NUM }</td>
</tr>
</tbody>
</table>

(G6)

\[
S ::= E$

\[
E ::= T E'$

\[
E' ::= + T E' \mid - T E' \mid$

\[
T ::= F T'$

\[
T' ::= * F T'$ \mid / F T'$ \mid$

\[
F ::= NUM \mid ID \mid ( E )$

\]
## Predictive Parsing Table for G6

Table $[X, T] = \text{Set of productions}$

$X ::= Y_1 \ldots Y_k \text{ in Table } [X, T]$

if $T \text{ in } \text{FIRST}[Y_1 \ldots Y_k]$

or if $(T \text{ in } \text{FOLLOW}[X] \text{ and nullable}[Y_1 \ldots Y_k])$

<table>
<thead>
<tr>
<th>+</th>
<th>*</th>
<th>(</th>
<th>)</th>
<th>ID</th>
<th>NUM</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>S ::= E$</td>
<td>S ::= E$</td>
<td>S ::= E$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>E ::= T E’</td>
<td>E ::= T E’</td>
<td>E ::= T E’</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E’</td>
<td>E’ ::= + T E’</td>
<td>E’ ::=</td>
<td>E’ ::=</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T ::= F T’</td>
<td>T ::= F T’</td>
<td>T ::= F T’</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T’</td>
<td>T’ ::=</td>
<td>T’ ::= * F T’</td>
<td>T’ ::=</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>F ::= (E)</td>
<td>F ::= ID</td>
<td>F ::= NUM</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(Entries for /, - are similar...)
Left-most derivation is constructed by recursive descent

(G6)
S ::= E$
E ::= T E'
E' ::= + T E' | − T E' | 
T ::= F T'
T' ::= * F T' | / F T' | 
F ::= NUM | ID | ( E )

Left-most derivation

S → E$
  → T E'$
  → F T' E'$
  → ( E ) T' E'$
  → ( T E' ) T' E'$
  → ( F T' E' ) T' E'$
  → ( 17 T' E' ) T' E'$
  → ( 17 E' ) T' E'$
  → ( 17 + T E' ) T' E'$
  → ( 17 + F T' E' ) T' E'$
  → ( 17 + 4 T' E' ) T' E'$
  → ( 17 + 4 E' ) T' E'$
  → ( 17 + 4 ) T' E'$
  → ( 17 + 4 ) * F T' E'$
  → ...
  → ...
  → ( 17 + 4 ) * ( 2 − 10 ) T' E'$
  → ( 17 + 4 ) * ( 2 − 10 ) E'$
  → ( 17 + 4 ) * ( 2 − 10 )

call S()
on ' ( ' call E()
on ' ( ' call T()
...
...

19
As a stack machine

\[
S \rightarrow E$

\rightarrow T E'$

\rightarrow F T' E'

\rightarrow (E') T' E'

\rightarrow (T E') T' E'

\rightarrow (F T' E') T' E'

\rightarrow (17 T' E') T' E'

\rightarrow (17 E') T' E'

\rightarrow (17 + T E') T' E'

\rightarrow (17 + F T' E') T' E'

\rightarrow (17 + 4 T' E') T' E'

\rightarrow (17 + 4 E') T' E'

\rightarrow (17 + 4) T' E'

\rightarrow (17 + 4) * F T' E'

\rightarrow ...

\rightarrow ...

\rightarrow (17 + 4) * (2 - 10) T' E'

\rightarrow (17 + 4) * (2 - 10) E'

\rightarrow (17 + 4) * (2 - 10)

E$

\rightarrow T E'

\rightarrow F T' E'

\rightarrow (E') T' E'

\rightarrow (T E') T' E'

\rightarrow (F T' E') T' E'

\rightarrow (17 T' E') T' E'

\rightarrow (17 E') T' E'

\rightarrow (17 + T E') T' E'

\rightarrow (17 + F T' E') T' E'

\rightarrow (17 + 4 T' E') T' E'

\rightarrow (17 + 4 E') T' E'

\rightarrow (17 + 4) T' E'

\rightarrow (17 + 4) * F T' E'

\rightarrow ...

\rightarrow ...

\rightarrow (17 + 4) * (2 - 10) T' E'

\rightarrow (17 + 4) * (2 - 10) E'

\rightarrow (17 + 4) * (2 - 10)
But wait! What if there are conflicts in the predictive parsing table?

### (G7)

<table>
<thead>
<tr>
<th>\text{Nullable}</th>
<th>\text{FIRST}</th>
<th>\text{FOLLOW}</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>{ c,d,a }</td>
<td>{ }</td>
</tr>
<tr>
<td>true</td>
<td>{ c }</td>
<td>{ c,d,a }</td>
</tr>
<tr>
<td>true</td>
<td>{ c,a }</td>
<td>{ c, a,d }</td>
</tr>
</tbody>
</table>

The resulting “predictive” table is not so predictive....

<table>
<thead>
<tr>
<th>a</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ \text{S ::= X Y S} }</td>
<td>{ \text{S ::= X Y S} }</td>
<td>{ \text{S ::= X Y S, S ::= d} }</td>
</tr>
<tr>
<td>{ \text{Y ::= } }</td>
<td>{ \text{Y ::= , Y ::= c} }</td>
<td>{ \text{Y ::= } }</td>
</tr>
<tr>
<td>{ \text{X ::= a, X ::= Y} }</td>
<td>{ \text{X ::= Y} }</td>
<td>{ \text{X ::= Y} }</td>
</tr>
</tbody>
</table>
• LL(k) : (L)eft-to-right parse, (L)eft-most derivation, k-symbol lookahead. Based on looking at the next k tokens, an LL(k) parser must predict the next production. We have been looking at LL(1).

• LR(k) : (L)eft-to-right parse, (R)ight-most derivation, k-symbol lookahead. Postpone production selection until the entire right-hand-side has been seen (and as many as k symbols beyond).

• LALR(1) : A special subclass of LR(1).
Example

(G8)

S ::= S ; S | ID = E | print (L)
E ::= ID | NUM | E + E | (S, E)
L ::= E | L, E

To be consistent, I should write the following, but I won’t…

(G8)

S ::= S SEMI S | ID EQUAL E | PRINT LPAREN L RPAREN
E ::= ID | NUM | E PLUS E | LPAREN S COMMA E RPAREN
L ::= E | L COMMA E
A right-most derivation ...

(G8)

\[ S ::= S ; S \]
\[ S \rightarrow S ; S \]
\[ S \rightarrow S ; \text{ID} = E \]
\[ S \rightarrow S ; \text{ID} = E + E \]
\[ S \rightarrow S ; \text{ID} = E + ( S, E ) \]
\[ S \rightarrow S ; \text{ID} = E + ( S, \text{ID} ) \]
\[ S \rightarrow S ; \text{ID} = E + ( S, \text{d} ) \]
\[ S \rightarrow S ; \text{ID} = E + ( \text{ID} = E, \text{d} ) \]
\[ S \rightarrow S ; \text{ID} = E + ( \text{ID} = E + E, \text{d} ) \]
\[ S \rightarrow S ; \text{ID} = E + ( \text{ID} = E + E, \text{d} ) \]
\[ S \rightarrow S ; \text{ID} = E + ( \text{ID} = \text{NUM}, \text{d} ) \]
\[ S \rightarrow S ; \text{ID} = E + ( \text{ID} = \text{NUM} + 6, \text{d} ) \]
\[ S \rightarrow S ; \text{ID} = E + ( \text{ID} = 5 + 6, \text{d} ) \]
\[ S \rightarrow S ; \text{ID} = \text{E} + ( d = 5 + 6, \text{d} ) \]
\[ S \rightarrow S ; \text{ID} = \text{E} + ( d = 5 + 6, \text{d} ) \]
\[ S \rightarrow S ; \text{ID} = c + ( d = 5 + 6, \text{d} ) \]
\[ S \rightarrow S ; b = c + ( d = 5 + 6, \text{d} ) \]
\[ \text{ID} = \text{E} ; b = c + ( d = 5 + 6, \text{d} ) \]
\[ \text{ID} = \text{NUM} ; b = c + ( d = 5 + 6, \text{d} ) \]
\[ \text{ID} = 7 ; b = c + ( d = 5 + 6, \text{d} ) \]
\[ a = 7 ; b = c + ( d = 5 + 6, \text{d} ) \]
Now, turn it upside down ...

→ a = 7 ; b = c + ( d = 5 + 6, d )
→ ID = 7 ; b = c + ( d = 5 + 6, d )
→ ID = NUM; b = c + ( d = 5 + 6, d )
→ ID = E ; b = c + ( d = 5 + 6, d )
→ S ; b = c + ( d = 5 + 6, d )
→ S ; ID = c + ( d = 5 + 6, d )
→ S ; ID = ID + ( d = 5 + 6, d)
→ S ; ID = E + ( d = 5 + 6, d )
→ S ; ID = E + ( ID = 5 + 6, d )
→ S ; ID = E + ( ID = NUM + 6, d )
→ S ; ID = E + ( ID = E + 6, d )
→ S ; ID = E + ( ID = E + NUM, d )
→ S ; ID = E + ( ID = E + E, d )
→ S ; ID = E + ( ID = E, d )
→ S ; ID = E + ( S, d )
→ S ; ID = E + ( S, ID )
→ S ; ID = E + ( S, E )
→ S ; ID = E + E
→ S ; ID = E
→ S ; S
→ S
Now, slice it down the middle...

| a = 7 ; b = c + ( d = 5 + 6, d ) |
| ID |
| ID = NUM |
| ID = E |
| S |
| S ; ID = ID |
| S ; ID = E |
| S ; ID = E + ( ID |
| S ; ID = E + ( ID = NUM |
| S ; ID = E + ( ID = E |
| S ; ID = E + ( ID = E + E |
| S ; ID = E + ( S |
| S ; ID = E + ( S, ID |
| S ; ID = E + ( S, E ) |
| S ; ID = E + E |
| S ; ID = E |
| S ; S |
| A stack of terminals and non-terminals |
| The rest of the input string |
Now, add some actions. $s = \text{SHIFT}, \hspace{0.5cm} r = \text{REDUCE}$

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s$</td>
<td>$s$</td>
</tr>
<tr>
<td>$s, s$</td>
<td>$r E ::= \text{NUM}$</td>
</tr>
<tr>
<td>$r S ::= \text{ID} = E$</td>
<td>$s, s$</td>
</tr>
<tr>
<td>$s, s$</td>
<td>$r E ::= \text{ID}$</td>
</tr>
<tr>
<td>$s, s, s$</td>
<td>$r E ::= \text{NUM}$</td>
</tr>
<tr>
<td>$s, s$</td>
<td>$r E ::= \text{ID}$</td>
</tr>
<tr>
<td>$r E ::= E + E, s, s$</td>
<td>$s, r E ::= (S, E)$</td>
</tr>
<tr>
<td>$r S ::= \text{ID} = E$</td>
<td>$r E ::= E + E$</td>
</tr>
<tr>
<td>$r S ::= \text{ID} = E$</td>
<td>$r S ::= S ; S$</td>
</tr>
</tbody>
</table>

$SHIFT = \text{LEX} + \text{move token to stack}$
LL(k) vs. LR(k) reductions

\[ A \rightarrow \beta \Rightarrow w' \ (\beta \in (T \cup N), \ w' \in T) \]

<table>
<thead>
<tr>
<th>( LL(k) )</th>
<th>( LR(k) )</th>
</tr>
</thead>
</table>

- **LL(k)**:
  - Stack: \( w' \)
  - 1 token look ahead
  - (left-most symbol at top)

- **LR(k)**:
  - Stack: \( w' \)
  - 1 token look ahead
  - (right-most symbol at top)

The language of this Stack IS REGULAR!
Q: How do we know when to shift and when to reduce? A: Build a FSA from LR(0) Items!

\((G10)\)

\[
\begin{align*}
S & ::= A \$ \\
A & ::= (A) \\
& | ( ) \\
\end{align*}
\]

If

\[
X ::= \alpha \beta
\]

is a production, then

\[
X ::= \alpha \cdot \beta
\]

is an LR(0) item.

\[
\begin{align*}
S & ::= \cdot A \$ \\
S & ::= A \cdot \$ \\
A & ::= \cdot (A) \\
A & ::= ( \cdot A ) \\
A & ::= ( A \cdot ) \\
A & ::= ( A ) \cdot \\
A & ::= \cdot ( ) \\
A & ::= ( \cdot ) \\
A & ::= ( ) \cdot \\
\end{align*}
\]

LR(0) items indicate what is on the stack (to the left of the \(\cdot\)) and what is still in the input stream (to the right of the \(\cdot\)).
LR(k) states (non-deterministic)

The state

\[( A \rightarrow \alpha \cdot \beta, \ a_1 a_2 \cdots a_k )\]

should represent this situation:

Input: \[ \quad w' \quad \]
Stack: \[ \quad \alpha \quad \]

(right-most symbol at top)

with

\[\beta a_1 a_2 \cdots a_k \Rightarrow w'\]
Key idea behind LR(0) items

• If the “current state” contains the item 
  \[ A ::= \alpha \cdot c \beta \] and the current symbol in the input buffer is c
  – the state prompts parser to perform a shift action
  – next state will contain \[ A ::= \alpha \cdot c \cdot \beta \]

• If the “state” contains the item \[ A ::= \alpha \cdot \] 
  – the state prompts parser to perform a reduce action

• If the “state” contains the item \[ S ::= \alpha \cdot \$ \] 
  and the input buffer is empty
  – the state prompts parser to accept

• But How about \[ A ::= \alpha \cdot X \beta \] where X is a nonterminal?
The NFA for LR(0) items

• The transition of LR(0) items can be represented by an NFA, in which
  – 1. each LR(0) item is a state,
  – 2. there is a transition from item $A ::= \alpha \cdot c \beta$ to item $A ::= \alpha c \cdot \beta$ with label $c$, where $c$ is a terminal symbol
  – 3. there is an $\varepsilon$-transition from item $A ::= \alpha \cdot X \beta$ to $X ::= \cdot \gamma$, where $X$ is a non-terminal
  – 4. $S ::= \cdot A \$ is the start state
  – 5. $A ::= \alpha \cdot$ is a final state.
Example NFA for Items

\[ S ::= \cdot A \$ \quad S ::= A \cdot \$ \quad A ::= \cdot (A) \]
\[ A ::= (\cdot A) \quad A ::= (A \cdot) \quad A ::= (A) \cdot \]
\[ A ::= \cdot ( ) \quad A ::= ( ) \cdot \]
The DFA from LR(0) items

• After the NFA for LR(0) is constructed, the resulting DFA for LR(0) parsing can be obtained by the usual NFA2DFA construction.

• we thus require
  – \(\varepsilon\)-closure \(I\)
  – \text{move}(S, a)

Fixed Point Algorithm for Closure\(I\)

  – Every item in \(I\) is also an item in Closure\(I\)
  – If \(A ::= \alpha \cdot B \beta\) is in Closure\(I\) and \(B ::= \cdot \gamma\) is an item, then add \(B ::= \cdot \gamma\) to Closure\(I\)
  – Repeat until no more new items can be added to Closure\(I\)
Examples of Closure

\[
\text{Closure}\left(\{A ::= ( \cdot A )\}\right) = \begin{cases} A ::= ( \cdot A ) \\ A ::= \cdot (A) \\ A ::= \cdot ( ) \end{cases}
\]

- closure(\{S ::= \cdot A \$\})
  \[
  \begin{cases} S ::= \cdot A \$ \\ A ::= \cdot (A) \\ A ::= \cdot ( ) \end{cases}
  \]

\[
\begin{align*}
S & ::= \cdot A \$ \\
A & ::= \cdot (A) \\
A & ::= ( A \cdot ) \\
A & ::= ( A ) \cdot \\
A & ::= ( ) \cdot
\end{align*}
\]
Goto() of a set of items

• Goto finds the new state after consuming a grammar symbol while in the current state

• Algorithm for $\text{Goto}(I, X)$ where $I$ is a set of items

$$\text{Goto}(I, X) = \text{Closure}( \{ A ::= \alpha X \beta \mid A ::= \alpha \cdot X \beta \text{ in } I \} )$$

• goto is the new set obtained by “moving the dot” over $X$
Examples of Goto

• Goto \( \{ A ::= \cdot (A) \}, () \)

\[
\begin{align*}
A & ::= (\cdot A) \\
A & ::= \cdot (A) \\
A & ::= \cdot ( )
\end{align*}
\]

• Goto \( \{ A ::= (\cdot A) \}, A \)

\[
\begin{align*}
A & ::= (A \cdot )
\end{align*}
\]

S ::= \cdot A $  
S ::= A \cdot $  
A ::= \cdot (A)  
A ::= (A \cdot )  
A ::= (A ) \cdot  
A ::= \cdot ( )  
A ::= ( ) \cdot
Essentially the usual NFA2DFA construction!!
Let A be the start symbol and S a new start symbol.
Create a new rule $S ::= A \$$
Create the first state to be $\text{Closure}\{ S ::= \cdot A \}$
Pick a state $I$
  – for each item $A ::= \alpha \cdot X \beta$ in $I$
    • find $\text{Goto}(I, X)$
    • if $\text{Goto}(I, X)$ is not already a state, make one
    • Add an edge $X$ from state $I$ to $\text{Goto}(I, X)$ state
Repeat until no more additions possible
DFA Example
Creating the Parse Table(s)

(G10)
(1) S ::= A$

(2) A ::= (A)

(3) A ::= ()

<table>
<thead>
<tr>
<th>State</th>
<th>(</th>
<th>)</th>
<th>$</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0</td>
<td>shift to s2</td>
<td></td>
<td>goto s1</td>
<td></td>
</tr>
<tr>
<td>s1</td>
<td></td>
<td>accept</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s2</td>
<td>shift to s2</td>
<td>shift to s5</td>
<td>goto s3</td>
<td></td>
</tr>
<tr>
<td>s3</td>
<td></td>
<td>shift to s4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s4</td>
<td>reduce (2)</td>
<td>reduce (2)</td>
<td>reduce (2)</td>
<td></td>
</tr>
<tr>
<td>s5</td>
<td>reduce (3)</td>
<td>reduce (3)</td>
<td>reduce (3)</td>
<td></td>
</tr>
</tbody>
</table>
Use table and top-of-stack and input symbol to get action:

If action is

- **shift sn**: advance input one token, push sn on stack
- **reduce X ::= α**: pop stack $2^* \mid α \mid$ times (grammar symbols are paired with states). In the state now on top of stack, use goto table to get next state sn, push it on top of stack
- **accept**: stop and accept
- **error**: weep (actually, produce a good error message)
### Parsing, again...

<table>
<thead>
<tr>
<th>State</th>
<th>ACTION</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0</td>
<td>shift to s2</td>
<td>goto s1</td>
</tr>
<tr>
<td>s1</td>
<td>accept</td>
<td></td>
</tr>
<tr>
<td>s2</td>
<td>shift to s2</td>
<td>shift to s5</td>
</tr>
<tr>
<td>s3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s4</td>
<td>reduce (2)</td>
<td>reduce (2)</td>
</tr>
<tr>
<td>s5</td>
<td>reduce (3)</td>
<td>reduce (3)</td>
</tr>
</tbody>
</table>

**Example:**

```plaintext
s0                                    ((()))$                           shift s2
s0 ( s2                               () )$                           shift s2
s0 ( s2 ( s2                         )))$                           shift s5
s0 ( s2 ( s2 ) s5                     ) )$                           reduce A ::= ()
goto s3
s0 ( s2 A                             ) )$                           goto s3
s0 ( s2 A s3                          ) )$                           shift s4
s0 ( s2 A s3 ) s4                     $                               reduce A ::= (A)
s0 A                                  $                               goto s1
s0 A s1                               $                               ACCEPT!
```
LR Parsing Algorithm

Stack of states and grammar symbols

| Sm    | Ym    | Sm-1 | Ym-1 |

Input sequence: a1 ... a_i ... a_n $

Action Table:
- terminals and $
- four different actions

Goto Table:
- non-terminal
- each item is a state number

Output
Problem With LR(0) Parsing

• No lookahead
• Vulnerable to unnecessary conflicts
  – Shift/Reduce Conflicts (may reduce too soon in some cases)
  – Reduce/Reduce Conflicts
• Solutions:
  – LR(1) parsing - systematic lookahead
LR(1) Items

• An LR(1) item is a pair:

\[(X ::= \alpha \cdot \beta, a)\]

  – \(X ::= \alpha\beta\) is a production
  – \(a\) is a terminal (the lookahead terminal)
  – LR(1) means 1 lookahead terminal

• \([X ::= \alpha \cdot \beta, a]\) describes a context of the parser

  – We are trying to find an \(X\) followed by an \(a\), and
  – We have (at least) \(\alpha\) already on top of the stack
  – Thus we need to see next a prefix derived from \(\beta a\)
The Closure Operation

• Need to modify closure operation:

\[
\text{Closure(Items)} = \\
\text{repeat} \\
\text{for each } [X ::= \alpha . Y\beta, a] \text{ in Items} \\
\text{for each production } Y ::= \gamma \\
\text{for each } b \text{ in } \text{First}(\beta a) \\
\text{add } [Y ::= \gamma, b] \text{ to Items} \\
\text{until Items is unchanged}
\]
Constructing the Parsing DFA (2)

• A DFA state is a closed set of LR(1) items

• The start state contains \( (S' ::= .S$, dummy) \)

• A state that contains \([X ::= \alpha., b]\) is labeled with “reduce with \(X ::= \alpha\) on lookahead \(b\)”

• And now the transitions …
The DFA Transitions

• A state $s$ that contains $[X ::= \alpha.Y\beta, b]$ has a transition labeled $y$ to the state obtained from $\text{Transition}(s, Y)$
  – $Y$ can be a terminal or a non-terminal

Transition($s$, $Y$)

Items = {} 

for each $[X ::= \alpha.Y\beta, b]$ in $s$
  add $[X ! \alpha.Y.\beta, b]$ to Items

return $\text{Closure}(\text{Items})$
• Shift and goto as before
• Reduce
  – state I with item \((A \rightarrow \alpha.., z)\) gives a reduce \(A \rightarrow \alpha\)
    if \(z\) is the next character in the input.

• LR(1)-parse tables can be very large
Use tools (LEX, YACC) or write by hand?

Some problems with auto-generation tools:

- Often slow
- Hard to generate good error messages for compiler users
- Often need to tweak grammar, but tool messages can be very obscure

This has led many to write lexers/parsers by hand and declare **YACC IS DEAD!**

My opinion: no “right answer” --- tools will continue to improve (see Menhir for example), but demand for increased compiler speed will also increase …
Compiler Construction
Lent Term 2018

Part II : Lectures 7 – 12 (of 16)

Timothy G. Griffin
tgg22@cam.ac.uk

Computer Laboratory
University of Cambridge
Starting from a direct implementation of Slang/L3 semantics, we will **DERIVE** a Virtual Machine in a step-by-step manner. The correctness of each step is (more or less) easy to check.

Lecture 7: We make this leap using intuition.
Later we will understand it more formally…
1. Interpreter 0: The high-level “definitional” interpreter
   1. Slang/L3 values represented directly as OCaml values
   2. Recursive interpreter implements a denotational semantics
   3. The interpreter implicitly uses OCaml’s runtime stack

2. Interpreter 2: A high-level stack-oriented machine
   1. Makes the Ocaml runtime stack explicit
   2. Complex values pushed onto stacks
   3. One stack for values and environments
   4. One stack for instructions
   5. Heap used only for references
   6. Instructions have tree-like structure
Approaches to Mathematical Semantics

- Axiomatic: Meaning defined through logical specifications of behaviour.
  - Hoare Logic (Part II)
  - Separation Logic
- Operational: Meaning defined in terms of transition relations on states in an abstract machine.
  - Semantics (Part 1B)
- Denotational: Meaning is defined in terms of mathematical objects such as functions.
  - Denotational Semantics (Part II)
A denotational semantics for L3?

\( N = \text{set of integers} \quad B = \text{set of booleans} \quad A = \text{set of addresses} \)

\( I = \text{set of identifiers} \quad \text{Expr} = \text{set of L3 expressions} \)

\( E = \text{set of environments} = I \rightarrow V \quad S = \text{set of stores} = A \rightarrow V \)

\( V = \text{set of values} \)

\( \approx A \)

\( + N \)

\( + B \)

\( + \{ () \} \)

\( + V \times V \)

\( + (V + V) \)

\( + (V \times S) \rightarrow (V \times S) \)

\( M = \text{the meaning function} \)

\( M : (\text{Expr} \times E \times S) \rightarrow (V \times S) \)

Set of values \( V \) solves this “domain equation” (here + means disjoint union).

Solving such equations is where some difficult maths is required …
Our shabby OCaml approximation

\[
\begin{align*}
A &= \text{set of addresses} \\
S &= \text{set of stores} = A \rightarrow V \\
V &= \text{set of value} \\
&\approx A \\
&\quad + \mathbb{N} \\
&\quad + \mathbb{B} \\
&\quad + \{ () \} \\
&\quad + V \times V \\
&\quad + (V + V) \\
&\quad + (V \times S) \rightarrow (V \times S) \\
E &= \text{set of environments} = A \rightarrow V \\
M &= \text{the meaning function} \\
M : (\text{Expr} \times E \times S) \rightarrow (V \times S)
\end{align*}
\]
let rec interpret (e, env, store) =
  match e with
  | If(e1, e2, e3) ->
    let (v, store') = interpret(e1, env, store) in
    (match v with
     | BOOL true -> interpret(e2, env, store')
     | BOOL false -> interpret(e3, env, store')
     | v -> complain "runtime error. Expecting a boolean!")
  | Pair(e1, e2) ->
    let (v1, store1) = interpret(e1, env, store) in
    let (v2, store2) = interpret(e2, env, store1) in
    (PAIR(v1, v2), store2)
  | Fst e ->
    (match interpret(e, env, store) with
     | (PAIR(v1, _), store') -> (v1, store')
     | (v, _) -> complain "runtime error. Expecting a pair!")
  | Snd e ->
    (match interpret(e, env, store) with
     | (PAIR(_, v2), store') -> (v2, store')
     | (v, _) -> complain "runtime error. Expecting a pair!")
  | Inl e -> let (v, store') = interpret(e, env, store) in (INL v, store')
  | Inr e -> let (v, store') = interpret(e, env, store) in (INR v, store')
  :.
Tricky bits: Slang functions mapped to OCaml functions!

let rec interpret (e, env, store) =
  match e with
  :
  :
  | Lambda(x, e) -> (FUN (fun (v, s) -> interpret(e, update(env, (x, v)), s)), store)
  | App(e1, e2) -> (* I chose to evaluate argument first! *)
    let (v2, store1) = interpret(e2, env, store) in
    let (v1, store2) = interpret(e1, env, store1) in
      (match v1 with
        | FUN f -> f (v2, store2)
        | v -> complain "runtime error. Expecting a function!")
  | LetFun(f, (x, body), e) ->
    let new_env =
      update(env, (f, FUN (fun (v, s) -> interpret(body, update(env, (x, v)), s))))
    in interpret(e, new_env, store)
  | LetRecFun(f, (x, body), e) ->
    let rec new_env g = (* a recursive environment!!! *)
      if g = f then FUN (fun (v, s) -> interpret(body, update(new_env, (x, v)), s))
      else env g
    in interpret(e, new_env, store)

update : env * (var * value) -> env
let fun f (x) = x + 1
    fun g(y) = f(y+2)+2
    fun h(w) = g(w+1)+3
in
    h(h(17))
end

The run-time data structure is the call stack containing an activation record for each function invocation.

Execution
interpret is implicitly using Ocaml’s runtime stack

```ocaml
let rec interpret (e, env, store) =
  match e with
  | Integer n         -> (INT n, store)
  | Op(e1, op, e2)    ->
    let (v1, store1) = interpret(e1, env, store) in
    let (v2, store2) = interpret(e2, env, store1) in
    (do_oper(op, v1, v2), store2)
  ..
  ..
```

• Every invocation of interpret is building an activation record on Ocaml’s runtime stack.

• **We will now define interpreter 2 which makes this stack explicit**
Inpterp_2 data types

type address

type store = address -> value

and value =
  | REF of address
  | INT of int
  | BOOL of bool
  | UNIT
  | PAIR of value * value
  | INL of value
  | INR of value
  | FUN of ((value * store) -> (value * store))

and env = Ast.var -> value

and instruction =
  | PUSH of value
  | LOOKUP of var
  | UNARY of unary_oper
  | OPER of oper
  | ASSIGN
  | SWAP
  | POP
  | BIND of var
  | FST
  | SND
  | DEREF
  | APPLY
  | MK_PAIR
  | MK_INL
  | MK_INR
  | MK_REF
  | MK_CLOSURE of code
  | MK_REC of var * code
  | TEST of code * code
  | CASE of code * code
  | WHILE of code * code

Interp_0

Interp_2
and code = instruction list

and binding = var * value

and env = binding list

type env_or_value = EV of env | V of value

type env_value_stack = env_or_value list

type state = code * env_value_stack

val step : state -> state

val driver : state -> value

val compile : expr -> code

val interpret : expr -> value

The state is actually comprised of a heap --- a global array of values --- a pair of the form

(code, evn_value_stack)
type state = code * env_value_stack

val step : state -> state

let step = function
(* (code stack, value/env stack) -> (code stack, value/env stack) *)
| ((PUSH v) :: ds, evs) -> (ds, (V v) :: evs)
| (POP :: ds, s :: evs) -> (ds, evs)
| (SWAP :: ds, s1 :: s2 :: evs) -> (ds, s2 :: s1 :: evs)
| ((BIND x) :: ds, (V v) :: evs) -> (ds, EV([(x, v)]) :: evs)
| ((LOOKUP x) :: ds, evs) -> (ds, V(search(evs, x)) :: evs)
| ((UNARY op) :: ds, (V v) :: evs) -> (ds, V(do_unary(op, v)) :: evs)
| ((OPER op) :: ds, (V v2) :: (V v1) :: evs) -> (ds, V(do_oper(op, v1, v2)) :: evs)
| (MK_PAIR :: ds, (V v2) :: (V v1) :: evs) -> (ds, V(PAIR(v1, v2)) :: evs)
| (FST :: ds, V(PAIR (v, _))) :: evs) -> (ds, (V v) :: evs)
| (SND :: ds, V(PAIR (_, v))) :: evs) -> (ds, (V v) :: evs)
| (MK_INL :: ds, (V v) :: evs) -> (ds, V(INL v) :: evs)
| (MK_INR :: ds, (V v) :: evs) -> (ds, V(INR v) :: evs)
| (CASE (c1, c2) :: ds, V(INL v) :: evs) -> (c1 @ ds, (V v) :: evs)
| (CASE (_, c2) :: ds, V(INR v) :: evs) -> (c2 @ ds, (V v) :: evs)
| ((TEST(c1, c2)) :: ds, V(BOOL true) :: evs) -> (c1 @ ds, evs)
| ((TEST(c1, c2)) :: ds, V(BOOL false) :: evs) -> (c2 @ ds, evs)
| (ASSIGN :: ds, (V v) :: (V (REF a)) :: evs) -> (heap.(a) <- v; (ds, V(UNIT) :: evs))
| (DEREF :: ds, (V (REF a)) :: evs) -> (ds, V(heap.(a)) :: evs)
| (MK_REF :: ds, (V v) :: evs) -> let a = allocate () in (heap.(a) <- v; (ds, V(REF a) :: evs))
| ((WHILE(c1, c2)) :: ds, V(BOOL false) :: evs) -> (ds, evs)
| ((WHILE(c1, c2)) :: ds, V(BOOL true) :: evs) -> (c1 @ ([WHILE(c1, c2)] @ ds, evs)
| (MK_CLOSURE c) :: ds, evs) -> (ds, V(mk_fun(c, evs_to_env evs)) :: evs)
| (MK_REC(f, c) :: ds, evs) -> (ds, V(mk_rec(f, c, evs_to_env evs)) :: evs)
| (APPLY :: ds, V(CLOSURE (_, (c, env)))) :: (V v) :: evs) -> (c @ ds, (V v) :: (EV env) :: evs)
| state -> complain ("step : bad state = " ^ (string_of_state state) ^ "\n")
The driver. Correctness

(* val driver : state -> value *)
let rec driver state =
  match state with
  | ([], [V v]) -> v
  | _ -> driver (step state)

val compile : expr -> code

The idea: if \( e \) passes the front-end and
\[ \text{Interp}_0.\text{interpret} \ e = v \]
then
\[ \text{driver} (\text{compile} \ e, []) = v' \]
where \( v' \) (somehow) represents \( v \).
Implement inter_0 in interp_2

```ocaml
let rec interpret (e, env, store) =  
  match e with  
  | Pair(e1, e2) ->  
    let (v1, store1) = interpret(e1, env, store) in  
    let (v2, store2) = interpret(e2, env, store1) in (PAIR(v1, v2), store2)  
  | Fst e ->  
    (match interpret(e, env, store) with  
     | (PAIR (v1, _), store') -> (v1, store')  
     | (v, _) -> complain "runtime error. Expecting a pair!"

let step = function  
  | (MK_PAIR :: ds, (V v2) :: (V v1) :: evs) -> (ds, V(PAIR(v1, v2)) :: evs)  
  | (FST :: ds, V(PAIR (v, _)) :: evs) -> (ds, (V v) :: evs)

let rec compile = function  
  | Pair(e1, e2) -> (compile e1) @ (compile e2) @ [MK_PAIR]  
  | Fst e -> (compile e) @ [FST]
```

interp_0.ml

interp_2.ml
Implement inter_0 in interp_2

let rec interpret (e, env, store) =
  match e with
  | If(e1, e2, e3) ->
    let (v, store') = interpret(e1, env, store) in
    (match v with
      | BOOL true -> interpret(e2, env, store')
      | BOOL false -> interpret(e3, env, store')
      | v -> complain "runtime error. Expecting a boolean!")

let step = function
  | ((TEST(c1, c2)) :: ds, V(BOOL true) :: evs) -> (c1 @ ds, evs)
  | ((TEST(c1, c2)) :: ds, V(BOOL false) :: evs) -> (c2 @ ds, evs)

let rec compile = function
  | If(e1, e2, e3) -> (compile e1) @ [TEST(compile e2, compile e3)]
let rec interpret (e, env, store) =
  match e with
  | Lambda(x, e) -> (FUN (fun (v, s) -> interpret(e, update(env, (x, v)), s)), store)
  | App(e1, e2) -> (* I chose to evaluate argument first! *)
    let (v2, store1) = interpret(e2, env, store) in
      let (v1, store2) = interpret(e1, env, store1) in
        (match v1 with
          | FUN f -> f (v2, store2)
          | v -> complain "runtime error. Expecting a function!")

let step = function
  | (POP :: ds, s :: evs) -> (ds, evs)
  | (SWAP :: ds, s1 :: s2 :: evs) -> (ds, s2 :: s1 :: evs)
  | ((BIND x) :: ds, (V v) :: evs) -> (ds, EV([(x, v)]) :: evs)
  | ((MK_CLOSURE c) :: ds, evs) -> (ds, V(mk_fun(c, evs_to_env evs)) :: evs)
  | (APPLY :: ds, V(CLOSURE (_, (c, env))) :: (V v) :: evs) -> (c @ ds, (V v) :: (EV env) :: evs)

let rec compile = function
  | Lambda(x, e) -> [MK_CLOSURE((BIND x) :: (compile e) @ [SWAP; POP]])
  | App(e1, e2) -> (compile e2) @ (compile e1) @ [APPLY; SWAP; POP]

interp_0.ml
interp_2.ml
Example: Compiled code for rev_pair.slang

```plaintext
let rev_pair (p : int * int) : int * int  = (snd p, fst p) in
  rev_pair (21, 17)
end

MK_CLOSURE([BIND p; LOOKUP p; SND; LOOKUP p; FST; MK_PAIR; SWAP; POP]);
BIND rev_pair;
PUSH 21;
PUSH 17;
MK_PAIR;
LOOKUP rev_pair;
APPLY;
SWAP;
POP;
SWAP;
POP
```

DEMO TIME!!!
1. “Flatten” code into linear array
2. Add “code pointer” (cp) to machine state
3. New instructions: LABEL, GOTO, RETURN
4. “Compile away” conditionals and while loops
Linearise code

Interpreter 2 copies code on the code stack. We want to introduce one global array of instructions indexed by a code pointer (\(cp\)). At runtime the \(cp\) points at the next instruction to be executed.

This will require two new instructions:

LABEL L : Associate label L with this location in the code array

GOTO L : Set the \(cp\) to the code address associated with L
Compile conditionals, loops

**If**($e_1, e_2, e_3$)

- code for $e_1$
- TEST $k$
- code for $e_2$
- GOTO $m$
- $k$: code for $e_3$
- $m$:

**While**($e_1, e_2$)

- $m$: code for $e_1$
- TEST $k$
- code for $e_2$
- GOTO $m$
- $k$:
If \( ? = 0 \) Then 17 else 21 end

interp_2

PUSH UNIT;
UNARY READ;
PUSH 0;
OPER EQI;
TEST(
  [PUSH 17],
  [PUSH 21]
)

interp_3

PUSH UNIT;
UNARY READ;
PUSH 0;
OPER EQI;
TEST L0;
PUSH 17;
GOTO L1;
LABEL L0;
PUSH 21;
LABEL L1;
HALT

interp_3 (loaded)

0: PUSH UNIT;
1: UNARY READ;
2: PUSH 0;
3: OPER EQI;
4: TEST L0 = 7;
5: PUSH 17;
6: GOTO L1 = 9;
7: LABEL L0;
8: PUSH 21;
9: LABEL L1;
10: HALT

Symbolic code locations

Numeric code locations
Implement inter_2 in interp_3

let step = function
  | ((TEST(c1, c2)) :: ds, V(BOOL true) :: evs) -> (c1 @ ds, evs)
  | ((TEST(c1, c2)) :: ds, V(BOOL false) :: evs) -> (c2 @ ds, evs)

let step (cp, evs) =
  match (get_instruction cp, evs) with
  | (TEST (_, Some _), V(BOOL true) :: evs) -> (cp + 1, evs)
  | (TEST (_, Some i), V(BOOL false) :: evs) -> (i, evs)
  | (LABEL l, evs) -> (cp + 1, evs)
  | (GOTO (_, Some i), evs) -> (i, evs)

Code locations are represented as

("L", None) : not yet loaded (assigned numeric address)

("L", Some i) : label “L” has been assigned numeric address i
Tricky bits again!

```ocaml
let step = function
| (POP :: ds,       s :: evs) -> (ds, evs)
| (SWAP :: ds,      s1 :: s2 :: evs) -> (ds, s2 :: s1 :: evs)
| ((BIND x) :: ds,  (V v) :: evs) -> (ds, EV([(x, v)]) :: evs)
| ((MK_CLOSURE c) :: ds,  evs) -> (ds, V(mk_fun(c, evs_to_env evs)) :: evs)
| (APPLY :: ds,     V(CLOSURE(_, (c, env))) :: (V v) :: evs)
```

```ocaml
let step (cp, evs) =
match (get_instruction cp, evs) with
| (POP,       s :: evs) -> (cp + 1, evs)
| (SWAP,      s1 :: s2 :: evs) -> (cp + 1, s2 :: s1 :: evs)
| (BIND x,    (V v) :: evs) -> (cp + 1, EV([(x, v)]) :: evs)
| (MK_CLOSURE loc,  evs) -> (cp + 1, 
                          V(CLOSURE(loc, evs_to_env evs)) :: evs)
| (RETURN,    (V v) :: _ :: (RA i) :: evs) -> (i, (V v) :: evs)
| (APPLY,     V(CLOSURE(_, (c, env))) :: (V v) :: evs)
```

Note that in interp_2 the body of a closure is consumed from the code stack. But in interp_3 we need to save the return address on the stack (here i is the location of the closure’s code).
Tricky bits again!

let rec compile = function
| Lambda(x, e)  -> [MK_CLOSURE((BIND x) :: (compile e) @ [SWAP; POP]])
| App(e1, e2)   -> (compile e2) @ (compile e1) @ [APPLY; SWAP; POP]

let rec comp = function
| App(e1, e2)   ->
  let (defs1, c1) = comp e1 in
  let (defs2, c2) = comp e2 in
  (defs1 @ defs2, c2 @ c1 @ [APPLY])
| Lambda(x, e)  ->
  let (defs, c) = comp e in
  let f = new_label () in
  let def = [LABEL f ; BIND x] @ c @ [SWAP; POP; RETURN] in
  (def @ defs, [MK_CLOSURE((f, None))])

let compile e =
  let (defs, c) = comp e in
  c                    (* body of program *)
  @ [HALT]            (* stop the interpreter *)
  @ defs              (* function definitions *)
let step {cp, evs} =
  match (get_instruction cp, evs) with
  | (PUSH v,)              evs) -> (cp + 1, (V v) :: evs)
  | (POP,                  s :: (cp + 1, evs))
  | (SWAP,                 s1 :: s2 :: evs) -> (cp + 1, s2 :: s1 :: evs)
  | (BIND x,               (V v) :: evs) -> (cp + 1, EV{[(x, v)]} :: evs)
  | (LOOKUP x,             evs) -> (cp + 1, V(search(evs, x)) :: evs)
  | (UNARY op,             (V v) :: evs) -> (cp + 1, V(do Unary(op, v)) :: evs)
  | (OPER op,              (V v2) :: (V v1) :: evs) -> (cp + 1, V(do Oper(op, v1, v2)) :: evs)
  | (MK_PAIR,              (V v2) :: (V v1) :: evs) -> (cp + 1, V(PAIR(v1, v2)) :: evs)
  | (FST,                  V(PAIR (v, _)) :: evs) -> (cp + 1, (V v) :: evs)
  | (SNR,                  V(PAIR (_, v)) :: evs) -> (cp + 1, (V v) :: evs)
  | (MK_INL,               (V v) :: evs) -> (cp + 1, V(INL v) :: evs)
  | (MK_INR,               (V v) :: evs) -> (cp + 1, V(INR v) :: evs)
  | (CASE _, Some _,       V(INL v) :: evs) -> (cp + 1, (V v) :: evs)
  | (CASE _, Some i,       V(INR v) :: evs) -> (i, (V v) :: evs)
  | (TEST _, Some _,       V(BOOL true) :: evs) -> (cp + 1, evs)
  | (TEST _, Some i,       V(BOOL false) :: evs) -> (i, evs)
  | (ASSIGN, (V v) :: (V (REF a)) :: evs) -> (heap.(a) <- v; (cp + 1, V(UNIT) :: evs))
  | (DEREF, (V (REF a)) :: evs) -> (cp + 1, V(heap.(a)) :: evs)
  | (MK_REF, (V v) :: evs) -> let a = new_address () in (heap.(a) <- v;
                             (cp + 1, V(REF a) :: evs))
  | (MK_CLOSURE loc,       evs) -> (cp + 1, V(CLOSURE(loc, evs_to_env evs)) :: evs)
  | (APPLY, V(CLOSURE {_, Some i}, env)) :: (V v) :: evs) -> (i, (V v) :: (EV env) :: (RA (cp + 1)) :: evs)
(* new instructions *)
  | (RETURN, (V v) :: _ :: (RA i) :: evs) -> (i, (V v) :: evs)
  | (LABEL i, evs) -> (cp + 1, evs)
  | (HALT, evs) -> (cp, evs)
  | (GOTO _, Some i), evs) -> (i, evs)
  | _ -> complain ("step: bad state = " ^ (string_of_state (cp, evs)) ^ "\n")
Some observations

• A very clean machine!
• But it still has a very inefficient treatment of environments.
• Also, pushing complex values on the stack is not what most virtual machines do. In fact, we are still using OCaml’s runtime memory management to manipulate complex values.
Example: Compiled code for rev_pair.slang

```
let rev_pair (p : int * int) : int * int = (snd p, fst p)
in
  rev_pair (21, 17)
end
```

MK_CLOSURE(
  [BIND p; LOOKUP p; SND; LOOKUP p; FST; MK_PAIR; SWAP; POP]);
BIND rev_pair;
PUSH 21;
PUSH 17;
MK_PAIR;
LOOKUP rev_pair;
APPLY;
SWAP;
POP;
SWAP;
POP

MK_CLOSURE(rev_pair)
BIND rev_pair
PUSH 21
PUSH 17
MK_PAIR
LOOKUP rev_pair
APPLY
SWAP
POP
HALT

LABEL rev_pair
BIND p
LOOKUP p
SND
LOOKUP p
FST
MK_PAIR
SWAP
POP
RETURN

DEM0 TIME!!!
1. **First change**: Introduce an **addressable stack**.
2. Replace variable lookup by a (relative) location on the stack or heap determined at **compile time**.
3. Relative to what? A **frame pointer** (fp) pointing into the stack is needed to keep track of the current **activation record**.
4. **Second change**: Optimise the representation of closures so that they contain **only** the values associated with the **free variables** of the closure and a pointer to code.
5. **Third change**: Restrict values on stack to be simple (ints, bools, heap addresses, etc). Complex data is moved to the heap, leaving pointers into the heap on the stack.
6. How might things look different in a language without first-class functions? In a language with multiple arguments to function calls?
Jargon Virtual Machine

Need for \texttt{fp} to be explained soon …
The stack in interpreter 3

A stack in interpreter 3

(1, (2, 17))

Inl(inr(99))

:   :   :
 :   :   :

Stack elements in interpreter 3 are not of fixed size.

Virtual machines (JVM, etc) typically restrict stack elements to be of a fixed size.

We need to shift data from the high-level stack of interpreter 3 to a lower-level stack with fixed size elements.

Solution: put the data in the heap. Place pointers to the heap on the stack.

“All problems in computer science can be solved by another level of indirection, except of course for the problem of too many indirections.”

--- David Wheeler
Some stack elements represent pointers into the heap.
type instruction =
| PUSH of value
| LOOKUP of Ast.var
| UNARY of Ast.unary_oper
| OPER of Ast.oper
| ASSIGN
| SWAP
| POP
| BIND of Ast.var
| FST
| SND
| DEREF
| APPLY
| RETURN
| MK_PAIR
| MK_INL
| MK_INR
| MK_REF
| MK_CLOSURE of location
| TEST of location
| CASE of location
| GOTO of location
| LABEL of label
| HALT

(* modified *)

type instruction =
| PUSH of stack_item
| LOOKUP of value_path
| UNARY of Ast.unary_oper
| OPER of Ast.oper
| ASSIGN
| SWAP
| POP
| FST
| SND
| DEREF
| APPLY
| RETURN
| MK_PAIR
| MK_INL
| MK_INR
| MK_REF
| MK_CLOSURE of location * int
| TEST of location
| CASE of location
| GOTO of location
| LABEL of label
| HALT

(* modified *)

(* not needed *)
type value = | REF of address | INT of int | BOOL of bool | UNIT
| PAIR of value * value | INL of value | INR of value | CLOSURE of location * env

type env_or_value = | EV of env | V of value | RA of address

type env_value_stack = env_or_value list

Jargon VM

The headers will be essential for garbage collection!
MK_INR (MK_INL is similar)

In interpreter 3

(MK_INR, (V v) :: evs) → (cp + 1, V(INR(v)) :: evs)

Jargon VM

The stack before

<table>
<thead>
<tr>
<th>v</th>
<th>MK_INR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The stack after

<table>
<thead>
<tr>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Newly allocated locations in the heap

<table>
<thead>
<tr>
<th>a: Header 2, INR</th>
</tr>
</thead>
<tbody>
<tr>
<td>a+1: v</td>
</tr>
</tbody>
</table>

Note: The header types are not really required. We could instead add an extra field here (for example, 0 or 1). However, header types aid in understanding the code and traces of runtime execution.
CASE (TEST is similar)

(CASE (_, Some _), V(INL v)::evs) -> (cp + 1, (V v) :: evs)
(CASE (_, Some i), V(INR v)::evs) -> (i, (V v) :: evs)
MK_PAIR

In interpreter 3:

\[(\text{MK\_PAIR}, (V v2) :: (V v1) :: \text{evs}) \rightarrow (\text{cp} + 1, V(\text{PAIR}(v1, v2)) :: \text{evs})\]

In Jargon VM:

The stack before

\[
\begin{array}{c}
v2 \\
v1 \\
: \\
: \\
: \\
: \\
\end{array}
\]

The stack after

\[
\begin{array}{c}
a \\
: \\
: \\
: \\
: \\
\end{array}
\]

Newly allocated locations in the heap

\[
\begin{array}{c}
a : \\
a + 1 : \\
a + 2 : \\
\end{array}
\]

Header 3, PAIR

\[
\begin{array}{c}
v1 \\
v2 \\
\end{array}
\]
FST (similar for SND)

In interpreter 3:

\[(FST, V \text{PAIR}(v1, v2)) :: evs) \rightarrow (cp + 1, v1 :: evs)\]

In Jargon VM:

The stack before

\[
\begin{array}{c}
\text{a} \\
\vdots \\
\vdots \\
\vdots
\end{array}
\]

The stack after

\[
\begin{array}{c}
v1 \\
\vdots \\
\vdots \\
\vdots
\end{array}
\]

Somewhere in the heap

\[
\begin{array}{c}
a : \\
a+1 : v1 \\
a+2 : v2
\end{array}
\]

Note that v1 could be a simple value (int or bool), or another heap address.
In interpreter 3:

```ocaml
let step (cp, evs) =
  match (get_instruction cp, evs) with
  | (MK_CLOSURE loc, evs) -> (cp + 1, V(CLOSURE(loc, evs_to_env evs)) :: evs)
  | (APPLY, V(CLOSURE ((_, Some i), env)) :: (V v) :: evs) -> (i, (V v) :: (EV env) :: (RA (cp + 1)) :: evs)
  | (RETURN, (V v) :: _ :: (RA i) :: evs) -> (i, (V v) :: evs)
```
MK_CLOSURE(c, n)

\[ c = \text{code location of start of instructions for closure,} \]
\[ n = \text{number of free variables in the body of closure.} \]

Put values associated with **free variables** on stack, then construct the closure on the heap.
A stack frame

Currently executing code for the closure at heap address “a” after it was applied to argument v.
Interpreter 3:

\[
\text{(APPLY, } V(\text{CLOSURE } ((\_, \text{Some } i), \text{env})) :: (V \ v) :: \text{evs}) \\
\rightarrow (i, (V \ v) :: (EV \ \text{env}) :: (RA \ (cp + 1)) :: \text{evs})
\]
Interpreter 3:

\[(\text{RETURN}, \ (V \ v) :: _ :: (RA \ i) :: evs) \rightarrow (i, \ (V \ v) :: evs)\]
Suppose we are executing code associated with this closure. Then every free variable in the body of the closure can be found from the frame pointer \( fp \):

- **Formal parameter**: at stack location \( fp-2 \)
- **Other free variables**:
  - Follow heap pointer found at \( fp-1 \)
  - Each free variable can be associated with a fixed offset from this heap address
LOOKUP (HEAP_OFFSET k)

Interpreter 3:
(LOOKUP x, evs) -> (cp + 1, V(search(evs, x)) :: evs)

Jargon VM:

BEFORE

FREE

k+1

j

a

v

AFTER

FREE

vk

v
k+1

j

a

v1

vk

Header
LOOKUP (STACK_OFFSET -2)

Interpreter 3:

\[(\text{LOOKUP } x, \text{ evs}) \rightarrow (cp + 1, V(\text{search(evs, x)}) :: \text{ evs})\]

Jargon VM:

BEFORE

\[
\begin{align*}
\text{sp} & \rightarrow \text{FREE} \\
& \quad : : \\
& \quad k+1 \\
& \quad j \\
& \quad a \\
& \quad v \\
& : : \\
& : : \\
\end{align*}
\]

AFTER

\[
\begin{align*}
\text{fp} & \rightarrow \text{FREE} \\
& \quad : : \\
& \quad v \\
& \quad : : \\
& \quad k+1 \\
& \quad j \\
& \quad a \\
& \quad v \\
& : : \\
& : : \\
\end{align*}
\]

push argument value onto the stack

LOOKUP (STACK_OFFSET -2)
Oh, one problem

```
let rec comp = function

  | LetFun(f, (x, e1), e2) ->
    let (defs1, c1) = comp e1 in
    let (defs2, c2) = comp e2 in
    let def = [LABEL f; BIND x] @ c1 @ [SWAP; POP; RETURN] in
    (def @ defs1 @ defs2,
     [MK_CLOSURE((f, None)); BIND f] @ c2 @ [SWAP; POP])

Problem: Code c2 can be anything --- how are we going to find the closure for f when we need it? It has to be a fixed offset from a frame pointer --- we no longer scan the stack for bindings!

let rec comp vmap = function

  | LetFun(f, (x, e1), e2) -> comp vmap (App(Lambda(f, e2), Lambda(x, e1)))
```

Solution in Jargon VM
For recursive function calls, push current closure on to the stack.

Jargon VM:

For recursive function calls, push current closure on to the stack.
Example : Compiled code for rev_pair.slang

```
let rev_pair (p : int * int) : int * int  = (snd p, fst p)
in
  rev_pair (21, 17)
end

After the front-end, compile treats this as follows.

```
App(
  Lambda(
    "rev_pair",
    App(Var "rev_pair", Pair (Integer 21, Integer 17))),
  Lambda("p", Pair(Snd (Var "p"), Fst (Var "p"))))
```

Example: Compiled code for rev_pair.slang

```
App(
  Lambda("rev_pair",
    App(Var "rev_pair", Pair (Integer 21, Integer 17))),
  Lambda("p", Pair(Snd (Var "p"), Fst (Var "p"))))
```

- Make closure for second lambda
- Make closure for first lambda
- do application
- the end!
- code for first lambda, push 21
- push 17
- make the pair on the heap
- push closure for second lambda on stack
- apply first lambda
- return from first lambda
- code for second lambda, push arg on stack
- extract second part of pair
- push arg on stack again
- extract first part of pair
- construct a new pair
- return from second lambda
Example: trace of rev_pair.slang execution

Installed Code =
0: MK_CLOSURE(L1 = 11, 0)
1: MK_CLOSURE(L0 = 4, 0)
2: APPLY
3: HALT
4: LABEL L0
5: PUSH STACK_INT 21
6: PUSH STACK_INT 17
7: MK_PAIR
8: LOOKUP STACK_LOCATION-2
9: APPLY
10: RETURN
11: LABEL L1
12: LOOKUP STACK_LOCATION-2
13: SND
14: LOOKUP STACK_LOCATION-2
15: FST
16: MK_PAIR
17: RETURN

========== state 1 ==========
cp = 0 -> MK_CLOSURE(L1 = 11, 0)
fp = 0
Stack =
1: STACK_RA 0
0: STACK_FP 0

========== state 2 ==========
cp = 1 -> MK_CLOSURE(L0 = 4, 0)
fp = 0
Stack =
2: STACK_HI 0
1: STACK_RA 0
0: STACK_FP 0

Heap =
0 -> HEAP_HEADER(2, HT_CLOSURE)
1 -> HEAP_CI 11

......
Example: trace of rev_pair.slang execution

====== state 15 ======

cp = 16 -> MK_PAIR  
fp = 8  
Stack =  
11: STACK_INT 21  
10: STACK_INT 17  
9: STACK_RA 10  
8: STACK_FP 4  
7: STACK_HI 0  
6: STACK_HI 4  
5: STACK_RA 3  
4: STACK_FP 0  
3: STACK_HI 2  
2: STACK_HI 0  
1: STACK_RA 0  
0: STACK_FP 0

Heap =  
0 -> HEAP_HEADER(2, HT_CLOSURE)  
1 -> HEAP_CI 11  
2 -> HEAP_HEADER(2, HT_CLOSURE)  
3 -> HEAP_CI 4  
4 -> HEAP_HEADER(3, HT_PAIR)  
5 -> HEAP_INT 21  
6 -> HEAP_INT 17  
7 -> HEAP_HEADER(3, HT_PAIR)  
8 -> HEAP_INT 17  
9 -> HEAP_INT 21

====== state 19 ======

cp = 3 -> HALT  
fp = 0  
Stack =  
2: STACK_HI 7  
1: STACK_RA 0  
0: STACK_FP 0

Heap =  
0 -> HEAP_HEADER(2, HT_CLOSURE)  
1 -> HEAP_CI 11  
2 -> HEAP_HEADER(2, HT_CLOSURE)  
3 -> HEAP_CI 4  
4 -> HEAP_HEADER(3, HT_PAIR)  
5 -> HEAP_INT 21  
6 -> HEAP_INT 17  
7 -> HEAP_HEADER(3, HT_PAIR)  
8 -> HEAP_INT 17  
9 -> HEAP_INT 21

Jargon VM:  
output> (17, 21)
Example: closure_add.slang

```haskell
let f(y : int) : int -> int = let g(x : int) : int = y + x in g end
in let add21 : int -> int = f(21)
    in let add17 : int -> int = f(17)
        in add17(3) + add21(10)
    end
end
```

Note: we really do need closures on the heap here — the values 21 and 17 do not exist on the stack at this point in the execution.

After the front-end, this becomes represented as follows.

```haskell
App(Lambda(f, App(Lambda(add21, 
    App(Lambda(add17, 
        Op(App(Var(add17), Integer(3)), 
            ADD, 
            App(Var(add21), Integer(10))))), 
        App(Var(f), Integer(17)))), 
    App(Var(f), Integer(21))))), 
Lambda(y, App(Lambda(g, Var(g)), Lambda(x, Op(Var(y), ADD, Var(x)))))
```
Can we make sense of this?

MK_CLOSURE(L3, 0)
MK_CLOSURE(L0, 0)
APPLY
HALT

L0:
  PUSH STACK_INT 21
LOOKUP STACK_LOCATION -2
APPLY
LOOKUP STACK_LOCATION -2
MK_CLOSURE(L1, 1)
APPLY
RETURN

L1:
  PUSH STACK_INT 17
LOOKUP HEAP_LOCATION 1
APPLY
LOOKUP STACK_LOCATION -2
MK_CLOSURE(L2, 1)
APPLY
RETURN

L2:
  PUSH STACK_INT 3
LOOKUP STACK_LOCATION -2
APPLY
PUSH STACK_INT 10
LOOKUP HEAP_LOCATION 1
APPLY
OPER ADD
RETURN

L3:
  LOOKUP STACK_LOCATION -2
MK_CLOSURE(L5, 1)
MK_CLOSURE(L4, 0)
APPLY
RETURN

L4:
  LOOKUP STACK_LOCATION -2
RETURN

L5:
  LOOKUP HEAP_LOCATION 1
LOOKUP STACK_LOCATION -2
OPER ADD
RETURN

144
The Gap, illustrated

let fib (m :int) : int =
  if m = 0
  then 1
  else if m = 1
    then 1
    else fib(m - 1) + fib(m - 2)
  end
end

slang.byte –c –i4 fib.slang

MK_CLOSURE(fib, 0)
MK_CLOSURE(L0, 0)
APPLY
HALT

L0 :
  PUSH STACK_UNIT
  UNARY READ
  LOOKUP STACK_LOCATION -2
  APPLY
  RETURN

fib :
  LOOKUP STACK_LOCATION -2
  PUSH STACK_INT 0
  OPER EQI
  TEST L1
  PUSH STACK_INT 1
  GOTO L2

L1 :
  LOOKUP STACK_LOCATION -2
  PUSH STACK_INT 1
  OPER EQI
  TEST L3
  PUSH STACK_INT 1
  GOTO L4

L3 :
  LOOKUP STACK_LOCATION -2
  PUSH STACK_INT 1
  OPER SUB
  LOOKUP STACK_LOCATION -1
  APPLY
  LOOKUP STACK_LOCATION -2
  PUSH STACK_INT 2
  OPER SUB
  LOOKUP STACK_LOCATION -1
  APPLY
  OPER ADD

L4 :
L2 :
  RETURN

Jargon VM code
1. The semantic GAP between a Slang/L3 program and a low-level translation (say x86/Unix) has been significantly reduced.


3. However, using a lower-level implementation (say x86, exploiting fast registers) to generate very efficient code is not so easy. See Part II Optimising Compilers.

Verification of compilers is an active area of research. See CompCert, CakeML, and DeepSpec.
What about languages other than Slang/L3?

• Many textbooks on compilers treat only languages with first-order functions --- that is, functions cannot be passes as an argument or returned as a result. In this case, we can avoid allocating environments on the heap since all values associated with free variables will be somewhere on the stack!

• But how do we find these values? We optimise stack search by following a chain of static links. Static links are added to every stack frame and the point to the stack frame of the last invocation of the defining function.

• One other thing: most languages take multiple arguments for a function/procedure call.
fun f(x, y) = e1

...  

fun g(w, v) = w + f(v, v)

For this invocation of the function f, we say that g is the caller while f is the callee.

Recursive functions can play both roles at the same time...
fun b(z) = e

fun g(x1) =
    fun h(x2) =
        fun f(x3) = e3(x1, x2, x3, b, g h, f)
        in
        e2(x1, x2, b, g, h, f)
    end
    in
    e1(x1, b, g, h)
end

...

b(g(17))
...

Pseudo-code
Function $g$ is the **definer** of $h$. Functions $g$ and $b$ must share a definer defined at depth $k-1$.
Stack with static links and variable number of arguments

The static link points down to the closest frame of definer at nesting depth \(i - 1\).

stack frame for **callee** defined at nesting depth \(i \leq k + 1\)

stack frame for **caller** defined at nesting **depth** \(k\) used to evaluate code at **depth** \(k + 1\).
caller and callee at same nesting depth $k$

call $f_0$

```
cp
j: call f
f: .........
```

```
cp
f: .........
```

```
sp
FREE
SL{k - 1}
j+1
```

```
fp
FREE
SL{k - 1}
```

caller’s frame
caller at depth \( k \) and callee at depth \( i < k \)

\[ \text{call } f(k - i) \]

```
fp := !(fp + 2);
for c = 1 to k - i
    p := !(p + 2);

SL{i-1} := p;
```
caller at depth k and callee at depth k + 1

call f (-1)

Code

j : call f

FREE

sp

SL\{k - 1\}

fp

FREE

FP-saved

j+1

FP-saved

SL\{k - 1\}
Access to argument values at static distance 0

sp → FREE

fp → SL
ra
V
fp - j → V

arg 0 j

sp → FREE
V
SL
ra
fp → FREE
V
SL
ra
Access to argument values at static distance $d$, $0 < d$

$p := !(fp + 2)$;
for $c = 1$ to $d$
{
    $p := !(p + 2)$;
}
$v := !(p - j)$;
What about Interpreter 1?

• Evaluation using a stack
• Recursion using a stack
• Tail recursion elimination: from recursion to iteration
• Continuation Passing Style (CPS): transform any recursive function to a tail-recursive function
• “Defunctionalisation” (DFC): replace higher-order functions with a data structure
• Putting it all together:
  – Derive the Fibonacci Machine
  – Derive the Expression Machine, and “compiler”!
• This provides a roadmap for the interp_0 \(\rightarrow\) interp_1 \(\rightarrow\) interp_2 derivations.
Example of tail-recursion: gcd

(* gcd : int * int -> int *)
let rec gcd(m, n) =
    if m = n
    then m
    else if m < n
        then gcd(m, n - m)
    else gcd(m - n, n)

Compared to fib, this function uses recursion in a different way. It is **tail-recursive**. If implemented with a stack, then the “call stack” (at least with respect to gcd) will simply grow and then shrink. No “ups and downs” in between.

Tail-recursive code can be replaced by iterative code that does not require a “call stack” (constant space)
Here we have illustrated tail-recursion elimination as a source-to-source transformation. However, the OCaml compiler will do something similar to a lower-level intermediate representation. Upshot: we will consider all tail-recursive OCaml functions as representing iterative programs.
Familiar examples: fold_left, fold_right

From ocaml-4.01.0/stdlib/list.ml:

(* fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

  fold_left f a [b1; ...; bn]] = f (... (f (f a b1) b2) ...) bn *
*)

let rec fold_left f a l =
  match l with
  | [] -> a
  | b :: rest -> fold_left f (f a b) rest

(* fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b

  fold_right f [a1; ...; an] b = f a1 (f a2 (... (f an b) ...)) *
*)

let rec fold_right f l b =
  match l with
  | [] -> b
  | a::rest -> f a (fold_right f rest b)
Question: can we transform any recursive function into a tail recursive function?

The answer is YES!

- We add an extra argument, called a *continuation*, that represents “the rest of the computation”
- This is called the Continuation Passing Style (CPS) transformation.
- We will then “defunctionalize” (DFC) these continuations and represent them with a stack.
- Finally, we obtain a tail recursive function that carries its own stack as an extra argument!

We will apply this kind of transformation to the code of interpreter 0 as the first steps towards deriving interpreter 1.
(CPS) transformation of fib

(* fib : int -> int *)
let rec fib m =
    if m = 0 then 1
    else if m = 1 then 1
    else fib(m - 1) + fib(m - 2)

(* fib_cps : int * (int -> int) -> int *)
let rec fib_cps (m, cnt) =
    if m = 0 then cnt 1
    else if m = 1 then cnt 1
    else fib_cps(m - 1, fun a -> fib_cps(m - 2, fun b -> cnt (a + b)))
The rest of the computation after computing “fib(m)”. That is, cnt is a function expecting the result of “fib(m)” as its argument.

\[
\text{let rec } \text{fib\_cps} (m, \text{ cnt}) = \\
\quad \text{if } m = 0 \text{ then } \text{ cnt } 1 \\
\quad \text{else if } m = 1 \text{ then } \text{ cnt } 1 \\
\quad \text{else } \text{ fib\_cps}(m - 1, \text{ fun } a \rightarrow \text{ fib\_cps}(m - 2, \text{ fun } b \rightarrow \text{ cnt } (a + b)))
\]

This makes explicit the order of evaluation that is implicit in the original “fib(m-1) + fib(m-2)”:

-- first compute fib(m-1)
-- then compute fib(m-1)
-- then add results together
-- then return
Expressed with “let” rather than “fun”

(* fib(cps)_v2 : (int -> int) * int -> int *)
let rec fib(cps)_v2 (m, cnt) =
  if m = 0
  then cnt 1
  else if m = 1
    then cnt 1
    else let cnt2 a b = cnt (a + b)
      in let cnt1 a = fib(cps)_v2(m - 2, cnt2 a)
      in fib(cps)_v2(m - 1, cnt1)

Some prefer writing CPS forms without explicit funs ....
Use the identity continuation ...

(* fib_cps : int * (int -> int) -> int *)

let rec fib_cps (m, cnt) =
    if m = 0
    then cnt 1
    else if m = 1
        then cnt 1
        else fib_cps(m - 1, fun a -> fib_cps(m - 2, fun b -> cnt (a + b)))

let id (x : int) = x

let fib_1 x = fib_cps(x, id)

List.map fib_1 [0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10];

= [1; 1; 2; 3; 5; 8; 13; 21; 34; 55; 89]
Correctness?

For all \( c : \text{int} \rightarrow \text{int} \), for all \( m, 0 \leq m \), we have, \( c(\text{fib } m) = \text{fib}_\text{cps}(m, c) \).

Proof: assume \( c : \text{int} \rightarrow \text{int} \). By Induction on \( m \). Base case: \( m = 0 \):

\[
\text{fib}_\text{cps}(0, c) = c(1) = c(\text{fib}(0)).
\]

Induction step: Assume for all \( n < m \), \( c(\text{fib } n) = \text{fib}_\text{cps}(n, c) \).
(That is, we need course-of-values induction!)

\[
\text{fib}_\text{cps}(m + 1, c)
= \begin{cases} 
  c(1) & \text{if } m + 1 = 1 \\
  \text{fib}_\text{cps}((m+1) -1, \text{fun } a \rightarrow \text{fib}_\text{cps}((m+1) -2, \text{fun } b \rightarrow c(a+b))) & \text{else}
\end{cases}
\]

\[
= \begin{cases} 
  c(1) & \text{if } m + 1 = 1 \\
  \text{fib}_\text{cps}(m, \text{fun } a \rightarrow \text{fib}_\text{cps}(m-1, \text{fun } b \rightarrow c(a+b))) & \text{else}
\end{cases}
\]

\[
= \begin{cases} 
  c(1) & \text{if } m + 1 = 1 \\
  (\text{by induction}) & \text{else}
\end{cases}
\]

\[
= \text{fib}_\text{cps}(m -1, \text{fun } b \rightarrow c(a+b)) \ (\text{fib } m)
\]

NB: This proof pretends that we can treat OCaml functions as ideal mathematical functions, which of course we cannot. OCaml functions might raise exceptions like "stack overflow" or "you burned my toast", and so on. But this is a convenient fiction as long as we remember to be careful.
Correctness?

= if m + 1 = 1
  then c 1
  else fib_cps(m-1, fun b -> c ((fib m) + b))
= (by induction)
  if m + 1 = 1
  then c 1
  else (fun b -> c ((fib m) + b)) (fib (m-1))
= if m + 1 = 1
  then c 1
  else c ((fib m) + (fib (m-1)))
= c (if m + 1 = 1
    then 1
    else ((fib m) + (fib (m-1))))
= c (if m + 1 = 1
    then 1
    else fib((m + 1) - 1) + fib ((m + 1) - 2))
= c (fib(m + 1))

QED.
Can with express fib_cps without a functional argument?

(* fib_cps_v2 : (int -> int) * int -> int *)
let rec fib_cps_v2 (m, cnt) =
    if m = 0
    then cnt 1
    else if m = 1
        then cnt 1
    else let cnt2 a b = cnt (a + b)
        in let cnt1 a = fib_cps_v2(m - 2, cnt2 a)
        in fib_cps_v2(m - 1, cnt1)

Idea of “defunctionalisation” (DFC): replace id, cnt1 and cnt2 with instances of a new data type:

type cnt = ID | CNT1 of int * cnt | CNT2 of int * cnt

Now we need an “apply” function of type  cnt * int -> int
(* datatype to represent continuations *)
type cnt = ID | CNT1 of int * cnt | CNT2 of int * cnt

(* apply_cnt : cnt * int -> int *)
let rec apply_cnt = function
| (ID, a) -> a
| (CNT1 (m, cnt), a) -> fib_cps_dfc(m - 2, CNT2 (a, cnt))
| (CNT2 (a, cnt), b) -> apply_cnt (cnt, a + b)

(* fib_cps_dfc : (cnt * int) -> int *)
and fib_cps_dfc (m, cnt) =
  if m = 0
  then apply_cnt(cnt, 1)
  else if m = 1
      then apply_cnt(cnt, 1)
      else fib_cps_dfc(m -1, CNT1(m, cnt))

(* fib_2 : int -> int *)
let fib_2 m = fib_cps_dfc(m, ID)
Let \(< c >\) be of type \(\text{cnt}\) representing a continuation \(c : \text{int} -\rightarrow \text{int}\) constructed by \(\text{fib}_\text{cps}\).

Then

\[
\text{apply}_\text{cnt}(< c >, m) = c(m)
\]
and

\[
\text{fib}_\text{cps}(n, c) = \text{fib}_\text{cps}_\text{dfc}(n, < c >).
\]

<table>
<thead>
<tr>
<th>Functional continuation (c)</th>
<th>Representation (&lt; c &gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{fun } a \rightarrow \text{fib}_\text{cps}(m - 2, \text{fun } b \rightarrow \text{cnt}(a + b)))</td>
<td>(\text{CNT1}(m, &lt; \text{cnt} &gt;))</td>
</tr>
<tr>
<td>(\text{fun } b \rightarrow \text{cnt}(a + b))</td>
<td>(\text{CNT2}(a, &lt; \text{cnt} &gt;))</td>
</tr>
<tr>
<td>(\text{fun } x \rightarrow x)</td>
<td>(\text{ID})</td>
</tr>
</tbody>
</table>
Eureka! Continuations are just lists (used like a stack)

\[
\text{type} \ \text{int\_list} = \ \text{NIL} \ | \ \text{CONS} \ \text{of} \ \text{int} \ \ast \ \text{int\_list}
\]

\[
\text{type} \ \text{cnt} = \ \text{ID} \ | \ \text{CNT1} \ \text{of} \ \text{int} \ \ast \ \text{cnt} \ | \ \text{CNT2} \ \text{of} \ \text{int} \ \ast \ \text{cnt}
\]

Replace the above continuations with lists! (I’ve selected more suggestive names for the constructors.)

\[
\text{type} \ \text{tag} = \ \text{SUB2} \ \text{of} \ \text{int} \ \ | \ \text{PLUS} \ \text{of} \ \text{int}
\]

\[
\text{type} \ \text{tag\_list\_cnt} = \ \text{tag} \ \text{list}
\]
The continuation lists are used like a stack!

```ocaml
type tag = SUB2 of int | PLUS of int

type tag_list_cnt = tag list

(* apply_tag_list_cnt : tag_list_cnt * int -> int *)
let rec apply_tag_list_cnt = function
  | ([], a) -> a
  | ((SUB2 m) :: cnt, a) -> fib_cps_dfc_tags(m - 2, (PLUS a):: cnt)
  | ((PLUS a) :: cnt, b) -> apply_tag_list_cnt (cnt, a + b)

(* fib_cps_dfc_tags : (tag_list_cnt * int) -> int *)
and fib_cps_dfc_tags (m, cnt) =
  if m = 0
  then apply_tag_list_cnt(cnt, 1)
  else if m = 1
  then apply_tag_list_cnt(cnt, 1)
  else fib_cps_dfc_tags(m - 1, (SUB2 m) :: cnt)

(* fib_3 : int -> int *)
let fib_3 m = fib_cps_dfc_tags(m, [])
```
Combine Mutually tail-recursive functions into a single function

```plaintext
type state_type =
   | SUB1 (* for right-hand-sides starting with fib_ *)
   | APPL (* for right-hand-sides starting with apply_ *)

type state = (state_type * int * tag_list_cnt) -> int

(* eval : state -> int * A two-state transition function *)
let rec eval = function
   | (SUB1, 0, cnt) -> eval (APPL, 1, cnt)
   | (SUB1, 1, cnt) -> eval (APPL, 1, cnt)
   | (SUB1, m, cnt) -> eval (SUB1, (m-1), (SUB2 m) :: cnt)
   | (APPL, a, (SUB2 m) :: cnt) -> eval (SUB1, (m-2), (PLUS a) :: cnt)
   | (APPL, b, (PLUS a) :: cnt) -> eval (APPL, (a+b), cnt)
   | (APPL, a, []) -> a

(* fib_4 : int -> int *)
let fib_4 m = eval (SUB1, m, [])
```

173
Eliminate tail recursion to obtain The Fibonacci Machine!

(* step : state -> state *)
let step = function
  | (SUB1, 0, cnt) -> (APPL, 1, cnt)
  | (SUB1, 1, cnt) -> (APPL, 1, cnt)
  | (SUB1, m, cnt) -> (SUB1, (m-1), (SUB2 m) :: cnt)
  | (APPL, a, (SUB2 m) :: cnt) -> (SUB1, (m-2), (PLUS a) :: cnt)
  | (APPL, b, (PLUS a) :: cnt) -> (APPL, (a+b), cnt)
  | _ -> failwith "step : runtime error!"

(* clearly TAIL RECURSIVE! *)
let rec driver state = function
  | (APPL, a, []) -> a
  | state -> driver (step state)

(* fib_5 : int -> int *)
let fib_5 m = driver (SUB1, m, [])

In this version we have simply made the tail-recursive structure very explicit.
Here is a trace of fib_5 6.

The OCaml file in basic_transformations/fibonacci_machine.ml contains some code for pretty printing such traces....
• What have we accomplished?
• We have taken a recursive function and turned it into an iterative function that does not require “stack space” for its evaluation (in OCaml)
• However, this function now carries its own evaluation stack as an extra argument!
• We have derived this iterative function in a step-by-step manner where each tiny step is easily proved correct.
• Wow!
type expr =
| INT of int
| PLUS of expr * expr
| SUBT of expr * expr
| MULT of expr * expr

This time we will derive a stack-machine AND a “compiler” that translates expressions into a list of instructions for the machine.

(* eval : expr -> int
  a simple recursive evaluator for expressions *)

let rec eval = function
| INT a -> a
| PLUS(e1, e2) -> (eval e1) + (eval e2)
| SUBT(e1, e2) -> (eval e1) - (eval e2)
| MULT(e1, e2) -> (eval e1) * (eval e2)
Here we go again : CPS

type cnt_2 = int -> int

type state_2 = expr * cnt_2

(* eval_aux_2 : state_2 -> int *)
let rec eval_aux_2 (e, cnt) =  
  match e with  
  | INT a -> cnt a  
  | PLUS(e1, e2) ->  
      eval_aux_2(e1, fun v1 -> eval_aux_2(e2, fun v2 -> cnt(v1 + v2)))  
  | SUBT(e1, e2) ->  
      eval_aux_2(e1, fun v1 -> eval_aux_2(e2, fun v2 -> cnt(v1 - v2)))  
  | MULT(e1, e2) ->  
      eval_aux_2(e1, fun v1 -> eval_aux_2(e2, fun v2 -> cnt(v1 * v2)))

(* id_cnt : cnt_2 *)
let id_cnt (x : int) = x

(* eval_2 : expr -> int *)
let eval_2 e = eval_aux_2(e, id_cnt)
Defunctionalise!

type cnt_3 =
  | ID
  | OUTER_PLUS of expr * cnt_3
  | OUTER_SUBT of expr * cnt_3
  | OUTER_MULT of expr * cnt_3
  | INNER_PLUS of int * cnt_3
  | INNER_SUBT of int * cnt_3
  | INNER_MULT of int * cnt_3

type state_3 = expr * cnt_3

(* apply_3 : cnt_3 * int -> int *)
let rec apply_3 = function
  | (ID, v) -> v
  | (OUTER_PLUS(e2, cnt), v1) -> eval_aux_3(e2, INNER_PLUS(v1, cnt))
  | (OUTER_SUBT(e2, cnt), v1) -> eval_aux_3(e2, INNER_SUBT(v1, cnt))
  | (OUTER_MULT(e2, cnt), v1) -> eval_aux_3(e2, INNER_MULT(v1, cnt))
  | (INNER_PLUS(v1, cnt), v2) -> apply_3(cnt, v1 + v2)
  | (INNER_SUBT(v1, cnt), v2) -> apply_3(cnt, v1 - v2)
  | (INNER_MULT(v1, cnt), v2) -> apply_3(cnt, v1 * v2)
(* eval_aux_2 : state_3 -> int *)
and eval_aux_3 (e, cnt) = 
  match e with 
  | INT a        -> apply_3(cnt, a) 
  | PLUS(e1, e2) -> eval_aux_3(e1, OUTER_PLUS(e2, cnt)) 
  | SUBT(e1, e2) -> eval_aux_3(e1, OUTER_SUBT(e2, cnt)) 
  | MULT(e1, e2) -> eval_aux_3(e1, OUTER_MULT(e2, cnt)) 

(* eval_3 : expr -> int *)
let eval_3 e = eval_aux_3(e, ID)
Eureka! Again we have a stack!

```ocaml
type tag =
    | O_PLUS of expr
    | I_PLUS of int
    | O_SUBT of expr
    | I_SUBT of int
    | O_MULT of expr
    | I_MULT of int

type cnt_4 = tag list

let rec apply_4 cnt v =
  | [] v -> v
  | O_PLUS e2 cnt v1 -> eval_aux_4(e2, I_PLUS v1 cnt)
  | O_SUBT e2 cnt v1 -> eval_aux_4(e2, I_SUBT v1 cnt)
  | O_MULT e2 cnt v1 -> eval_aux_4(e2, I_MULT v1 cnt)
  | I_PLUS v1 cnt v2 -> apply_4 cnt v1 + v2
  | I_SUBT v1 cnt v2 -> apply_4 cnt v1 - v2
  | I_MULT v1 cnt v2 -> apply_4 cnt v1 * v2
```

(* apply_4 : cnt_4 * int -> int *)
Eureka! Again we have a stack!

(* eval_aux_4 : state_4 -> int *)
and eval_aux_4 (e, cnt) =
  match e with
  | INT  a         -> apply_4(cnt, a)
  | PLUS(e1, e2)   -> eval_aux_4(e1, O_PLUS(e2) :: cnt)
  | SUBT(e1, e2)   -> eval_aux_4(e1, O_SUBT(e2) :: cnt)
  | MULT(e1, e2)   -> eval_aux_4(e1, O_MULT(e2) :: cnt)

(* eval_4 : expr -> int *)
let eval_4 e = eval_aux_4(e, [])
Eureka! Can combine apply_4 and eval_aux_4

Type of an “accumulator” that contains either an int or an expression.

```
type acc =
    | A_INT of int
    | A_EXP of expr

type cnt_5 = cnt_4

type state_5 = cnt_5 * acc

val : step : state_5 -> state_5

val driver : state_5 -> int

val eval_5 : expr -> int
```

The driver will be clearly tail-recursive …
let step_5 = function
| (cnt, A_EXP (INT a)) -> (cnt, A_INT a)
| (cnt, A_EXP (PLUS(e1, e2))) -> (O_PLUS(e2) :: cnt, A_EXP e1)
| (cnt, A_EXP (SUBT(e1, e2))) -> (O_SUBT(e2) :: cnt, A_EXP e1)
| (cnt, A_EXP (MULT(e1, e2))) -> (O_MULT(e2) :: cnt, A_EXP e1)
| ((O_PLUS e2) :: cnt, A_INT v1) -> ((I_PLUS v1) :: cnt, A_EXP e2)
| ((O_SUBT e2) :: cnt, A_INT v1) -> ((I_SUBT v1) :: cnt, A_EXP e2)
| ((O_MULT e2) :: cnt, A_INT v1) -> ((I_MULT v1) :: cnt, A_EXP e2)
| ((I_PLUS v1) :: cnt, A_INT v2) -> (cnt, A_INT (v1 + v2))
| ((I_SUBT v1) :: cnt, A_INT v2) -> (cnt, A_INT (v1 - v2))
| ((I_MULT v1) :: cnt, A_INT v2) -> (cnt, A_INT (v1 * v2))
| ([], A_INT v) -> ([], A_INT v)

let rec driver_5 = function
| ([], A_INT v) -> v
| state -> driver_5 (step_5 state)

let eval_5 e = driver_5([], A_EXP e)
Eureka! There are really two independent stacks here --- one for “expressions” and one for values

type directive =
  | E of expr
  | DO_PLUS
  | DO_SUBT
  | DO_MULT

type directive_stack = directive list

type value_stack = int list

val step_6 : state_6 -> state_6
val driver_6 : state_6 -> int
val exp_6 : expr -> int

The state is now two stacks!
let step_6 = function
| (E(INT v) :: ds, vs) -> (ds, v :: vs)
| (E(PLUS(e1, e2)) :: ds, vs) -> ((E e1) :: (E e2) :: DO_PLUS :: ds, vs)
| (E(SUBT(e1, e2)) :: ds, vs) -> ((E e1) :: (E e2) :: DO_SUBT :: ds, vs)
| (E(MULT(e1, e2)) :: ds, vs) -> ((E e1) :: (E e2) :: DO_MULT :: ds, vs)
| (DO_PLUS :: ds, v2 :: v1 :: vs) -> (ds, (v1 + v2) :: vs)
| (DO_SUBT :: ds, v2 :: v1 :: vs) -> (ds, (v1 - v2) :: vs)
| (DO_MULT :: ds, v2 :: v1 :: vs) -> (ds, (v1 * v2) :: vs)
| _ -> failwith "eval : runtime error!"

let rec driver_6 = function
  | ([], [v]) -> v
  | state     -> driver_6 (step_6 state)

let eval_6 e = driver_6 ([E e], [])
An eval_6 trace

e = PLUS(MULT(INT 89, INT 2), SUBT(INT 10, INT 4))
This evaluator is interleaving two distinct computations:

(1) decomposition of the input expression into sub-expressions
(2) the computation of +, -, and *

Idea: why not do the decomposition BEFORE the computation?

Key insight: An interpreter can (usually) be refactored into a translation (compilation!) followed by a lower-level interpreter.

\[
\text{Interpret\_higher} (e) = \text{interpret\_lower} (\text{compile}(e))
\]

Note: this can occur at many levels of abstraction: think of machine code being interpreted in micro-code …
Refactor --- compile!

(* low-level instructions *)

```ocaml
type instr =
  | Ipush of int
  | Iplus
  | Isubt
  | Imult

type code = instr list

type state_7 = code * value_stack

(* compile : expr -> code *)

let rec compile = function
  | INT a                  -> [Ipush a]
  | PLUS(e1, e2)   -> (compile e1) @ (compile e2) @ [Iplus]
  | SUBT(e1, e2)   -> (compile e1) @ (compile e2) @ [Isubt]
  | MULT(e1, e2)  -> (compile e1) @ (compile e2) @ [Imult]
```

Never put off till run-time what you can do at compile-time.
-- David Gries
Evaluate compiled code.

(* step_7 : state_7 -> state_7 *)

let step_7 = function
  | (lpush v :: is, vs) -> (is, v :: vs)
  | (lplus :: is, v2::v1::vs) -> (is, (v1 + v2) :: vs)
  | (lsubt :: is, v2::v1::vs) -> (is, (v1 - v2) :: vs)
  | (lmult :: is, v2::v1::vs) -> (is, (v1 * v2) :: vs)
  | _ -> failwith "eval : runtime error!"

let rec driver_7 = function
  | ([], [v]) -> v
  | _ -> driver_7 (step_7 state)

let eval_7 e = driver_7 (compile e, [])
An eval_7 trace

\[
\text{compile (PLUS(MULT(INT 89, INT 2), SUBT(INT 10, INT 4)))} = \text{[push 89; push 2; mult; push 10; push 4; subt; plus]}
\]

\[
\begin{align*}
\text{state 1} & \quad IS = \text{[add; sub; push 4; push 10; mul; push 2; push 89]} \\
& \quad VS = [] \\
\text{state 2} & \quad IS = \text{[add; sub; push 4; push 10; mul; push 2]} \\
& \quad VS = [89] \\
\text{state 3} & \quad IS = \text{[add; sub; push 4; push 10; mul]} \\
& \quad VS = [89; 2] \\
\text{state 4} & \quad IS = \text{[add; sub; push 4; push 10]} \\
& \quad VS = [178] \\
\text{state 5} & \quad IS = \text{[add; sub; push 4]} \\
& \quad VS = [178; 10] \\
\text{state 6} & \quad IS = \text{[add; sub]} \\
& \quad VS = [178; 10; 4] \\
\text{state 7} & \quad IS = \text{[add]} \\
& \quad VS = [178; 6] \\
\text{state 8} & \quad IS = [] \\
& \quad VS = [184]
\end{align*}
\]

Top of each stack is on the right
The derivation from eval to compile+eval_7 can be used as a guide to a derivation from Interpreter 0 to interpreter 2.

1. Apply CPS to the code of Interpreter 0
2. Defunctionalise
3. Arrive at interpreter 1, which has a single continuation stack containing expressions, values and environments
4. Spit this stack into two stacks : one for instructions and the other for values and environments
5. Refactor into compiler + lower-level interpreter
6. Arrive at interpreter 2.
Starting from a direct implementation of Slang/L3 semantics, we have **DERIVED** a Virtual Machine in a step-by-step manner. The correctness of each step is (more or less) easy to check.
Part III : Lectures 13 – 16

- 13 : Compilers in their OS context
- 14 : Assorted Topics
- 15 : Runtime memory management
- 16 : Bootstrapping a compiler

Timothy G. Griffin
tgg22@cam.ac.uk
Computer Laboratory
University of Cambridge
Lecture 13

- Code generation for multiple platforms.
- Assembly code
- Linking and loading
- The Application Binary Interface (ABI)
- Object file format (only ELF covered)
- A crash course in x86 architecture and instruction set
- Naïve generation of x86 code from Jargon VM instructions
We could implement a Jargon byte code interpreter ...

... void vsm_execute_instruction(vsm_state *state, bytecode instruction) {
  opcode code = instruction.code;
  argument arg1 = instruction.arg1;
  switch (code) {
    case PUSH: { state->stack[state->sp++] = arg1; state->pc++; break; }
    case POP: { state->sp--; state->pc++; break; }
    case GOTO: { state->pc = arg1; break; }
    case STACK_LOOKUP: {
      state->stack[state->sp++] =
        state->stack[state->fp + arg1];
      state->pc++; break; }
  ...
  ...
}
...
Backend could target multiple platforms

One of the great benefits of Virtual Machines is their portability. However, for more efficient code we may want to compile to assembler. Lost portability can be regained through the extra effort of implementing code generation for every desired target platform.
Assembly and Linking

From symbolic names and addresses to numeric codes and numeric addresses

Name resolution, create single address space by address relocation

(main tasks)
9 Binary Compatibility

Binary compatibility encompasses several related concepts:

`application binary interface (ABI)`

The set of runtime conventions followed by all of the tools that deal with binary representations of a program, including compilers, assemblers, linkers, and language runtime support. Some ABIs are formal with a written specification, possibly designed by multiple interested parties. Others are simply the way things are actually done by a particular set of tools.
Applications Binary Interface (ABI)

We will use x86/Unix as our running example. Specifies many things, including the following.

• C calling conventions used for systems calls or calls to compiled C code.
  • Register usage and stack frame layout
  • How parameters are passed, results returned
  • Caller/callee responsibilities for placement and cleanup
• Byte-level layout and semantics of object files.
  • Executable and Linkable Format (ELF). Formerly known as Extensible Linking Format.
• Linking, loading, and name mangling

Note: the conventions are required for portable interaction with compiled C. Your compiled language does not have to follow the same conventions!
Object files

Must contain at least

- Program instructions
- Symbols being exported
- Symbols being imported
- Constants used in the program (such as strings)

Executable and Linkable Format (ELF) is a common format for both linker input and output.
<table>
<thead>
<tr>
<th>Header information; positions and sizes of sections</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>.text</strong> segment (code segment): binary data</td>
</tr>
<tr>
<td><strong>.data</strong> segment: binary data</td>
</tr>
<tr>
<td><strong>.rela.text</strong> code segment relocation table: list of (offset,symbol) pairs giving:</td>
</tr>
<tr>
<td>(i) offset within <strong>.text</strong> to be relocated; and</td>
</tr>
<tr>
<td>(iii) by which symbol</td>
</tr>
<tr>
<td><strong>.rela.data</strong> data segment relocation table: list of (offset,symbol) pairs giving:</td>
</tr>
<tr>
<td>(i) offset within <strong>.data</strong> to be relocated; and</td>
</tr>
<tr>
<td>(iii) by which symbol</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
... 

**.symtab** symbol table:
List of external symbols (as triples) used by the module. Each is (attribute, offset, symname) with attribute:
1. **undef**: externally defined, offset is ignored;
2. **defined in code segment** (with offset of definition);
3. **defined in data segment** (with offset of definition).
Symbol names are given as offsets within **.strtab** to keep table entries of the same size.

**.strtab** string table:
the string form of all external names used in the module
What does a linker do?
• takes some object files as input, notes all undefined symbols.
• recursively searches libraries adding ELF files which define such symbols until all names defined (“library search”).
• whinges if any symbol is undefined or multiply defined.

Then what?
• concatenates all code segments (forming the output code segment).
• concatenates all data segments.
• performs relocations (updates code/data segments at specified offsets).
Dynamic vs. Static linking

Static linking (compile time)
Problem: a simple “hello world” program may give a 10MB executable if it refers to a big graphics or other library.

Dynamic linking (run time)
For shared libraries, the object files contain stubs, not code, and the operating system loads and links the code on demand.

Pros and Cons of dynamic linking:

(+ ) Executables are smaller
(+ ) Bug fixes to libraries don’t require re-linking.
(-) Non-compatible changes to a library can wreck previously working programs (“dependency hell”).
A “runtime system”

A library implementing functionality needed to run compiled code on a given operating system. Normally tailored to the language being compiled.

- Implements interface between OS and language.
- May implement memory management.
- May implement “foreign function” interface (say we want to call compiled C code from Slang code, or vice versa).
- May include efficient implementations of primitive operations defined in the compiled language.
- For some languages, the runtime system may perform runtime type checking, method lookup, security checks, and so on.
- ...
In either case, implementers of the compiler and the runtime system must agree on many low-level details of memory layout and data representation.
Rough schematic of traditional layout in (virtual) memory.

The heap is used for dynamically allocating memory. Typically either for very large objects or for those objects that are returned by functions/procedures and must outlive the associated activation record.

In languages like Java and ML, the heap is managed automatically ("garbage collection")
A CISC architecture
There are 16, 32 and 64 bit versions
32 bit version:
- General purpose registers: EAX EBX ECX EDX
- Special purpose registers: ESI EDI EBP EIP ESP
  - EBP: normally used as the frame pointer
  - ESP: normally used as the stack pointer
  - EDI: often used to pass (first) argument
  - EIP: the code pointer
- Segment and flag registers that we will ignore …
64 bit version:
- Rename 32-bit registers with “R” (RAX, RBX, RCX, …)
- More general registers: R8 R9 R10 R11 R12 R13 R14 R15

Register names can indicate “width” of a value.

rax: 64 bit version
eax: 32 bit version (or lower 32 bits of rax)
ax: 16 bit version (or lower 16 bits of eax)
al: lower 8 bits of ax
ah: upper 8 bits of ax
The syntax of x86 assembler comes in several flavours. Here are two examples of “put integer 4 into register eax”:

```
movl $4, %eax          // GAS (aka AT&T) notation
mov  eax, 4                // Intel notation
```

I will (mostly) use the GAS syntax, where a suffix is used to indicate width of arguments:

- `b` (byte) = 8 bits
- `w` (word) = 16 bits
- `l` (long) = 32 bits
- `q` (quad) = 64 bits

For example, we have `movb`, `movw`, `movl`, and `movq`. 
Examples (in GAS notation)

movl $4, %eax # put 32 bit integer 4 in register eax
movw $4, %eax # put 16 bit integer 4 in lower 16 bits of eax
movb $4, %eax # put 4 bit integer 4 in lowest 4 bits of eax
movl %esp, %ebp # put the contents of esp into ebp
movl (%esp), %ebp # interpret contents of esp as a memory
# address. Copy the value at that address
# into register ebp
movl %esp, (%ebp) # interpret contents of ebp as a memory
# address. Copy the value in esp to
# that address.
movl %esp, 4(%ebp) # interpret contents of ebp as a memory
# address. Add 4 to that address. Copy
# the value in esp to this new address.
A few more examples

```
call label  # push return address on stack and jump to label
ret        # pop return address off stack and jump there
# NOTE: managing other bits of the stack frame
# such as stack and frame pointer must be done
# explicitly
subl $4, %esp  # subtract 4 from esp. That is, adjust the
# stack pointer to make room for one 32-bit
# (4 byte) value. (stack grows downward!)
```

Assume that we have implemented a procedure in C called allocate that will manage heap memory. We will compile and link this in with code generated by the slang compiler. At the x86 level, allocate will expect a header in edi and return a heap pointer in eax.
Some Jargon VM instructions are “easy” to translate

Remember: X86 is CISC, so RISC architectures may require more instructions ...

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GOTO loc</td>
<td>jmp loc</td>
<td></td>
</tr>
<tr>
<td>POP</td>
<td>addl $4, %esp</td>
<td>// move stack pointer 1 word = 4 bytes</td>
</tr>
<tr>
<td>PUSH v</td>
<td>subl $4, %esp</td>
<td>// make room on top of stack</td>
</tr>
<tr>
<td></td>
<td>movl $i, (%esp)</td>
<td>// where i is an integer representing v</td>
</tr>
<tr>
<td>FST</td>
<td>movl (%esp), %edx</td>
<td>// store “a” into edx</td>
</tr>
<tr>
<td></td>
<td>movl 4(%edx), %edx</td>
<td>// load v1, 4 bytes, 1 word, after header</td>
</tr>
<tr>
<td></td>
<td>movl %edx, (%esp)</td>
<td>// replace “a” with “v1” at top of stack</td>
</tr>
<tr>
<td>SND</td>
<td>movl (%esp), %edx</td>
<td>// store “a” into edx</td>
</tr>
<tr>
<td></td>
<td>movl 8(%edx), %edx</td>
<td>// vload v2, 8 bytes, 2 words, after header</td>
</tr>
<tr>
<td></td>
<td>movl %edx, (%esp)</td>
<td>// replace “a” with “v2” at top of stack</td>
</tr>
</tbody>
</table>

![Diagram](https://via.placeholder.com/150)
... while others require more work

One possible x86 (32 bit) implementation of \texttt{MK\_PAIR}:

\begin{verbatim}
movl $3, %edi          // construct header in edi
shr $16, %edi,        // ... put size in upper 16 bits (shift right)
movw $PAIR, %di       // ... put type in lower 16 bits of edi
call allocate         // input: header in ebi, output: “a” in eax
movl (%esp), %edx     // move “v2” to the heap,
movl %edx, 8(%eax)    // ... using temporary register edx
addl $4, %esp         // adjust stack pointer (pop “v2”)
movl (%esp), %edx     // move “v1” to the heap
movl %edx, 4(%eax)    // ... using temporary register edx
movl %eax, (%esp)     // copy value “a” to top of stack
\end{verbatim}
Left as exercises for you:

Here’s a hint. For things you don’t understand, just experiment! OK, you need to pull an address out of a closure and call it. Hmm, how does something similar get compiled from C?

```
int func ( int (*f)(int) ) { return (*f)(17); } /* pass a function pointer and apply it */
```

X86, 64 bit without –O2

```plaintext
_func:
pushq   %rbp   # save frame pointer
movq    %rsp, %rbp # set frame pointer to stack pointer
subq    $16, %rsp # make some room on stack
movl    $17, %eax # put 17 in argument register eax
movq    %rdi, -8(%rbp) # rdi contains the argument f
movl    %eax, %edi # put 17 in register edi, so f will get it
callq   *-8(%rbp) # WOW, a computed address for call!
addq    $16, %rsp # restore stack pointer
popq    %rbp # restore old frame pointer
ret     # restore stack
```
What about arithmetic?

Houston, we have a problem….

- It may not be obvious now, but if we want to have automated memory management we need to be able to distinguish between values (say integers) and pointers at runtime.
- Have you ever noticed that integers in SML or Ocaml are either 31 (or 63) bits rather than the native 32 (or 64) bits?
  - That is because these compilers use a the least significant bit to distinguish integers (bit = 1) from pointers (bit = 0).
  - OK, this works. But it may complicate every arithmetic operation!
  - This is another exercise left for you to ponder…
1. Stacks are slow, registers are fast
   1. Stack frames still needed ...
   2. ... but try to shift work into registers
   3. Caller/callee save/restore policies
   4. Register spilling

2. Simple optimisations
   1. Peep hole (sliding window)
   2. Constant propagation
   3. Inlining

3. Representing objects (as in OOP)
   1. At first glance objects look like a closure containing multiple function (methods) ...
   2. ... but complications arise with method dispatch

4. Implementing exception handling on the stack
Stack vs registers

Stack-oriented:
(+): argument location is implicit, so instructions are smaller.
(---): Execution is slower.

Register-oriented:
(+++): Execution MUCH faster.
(-): argument location is explicit, so instructions are larger.
Main dilemma: registers are fast, but are fixed in number. And that number is rather small.

- Manipulating the stack involves RAM access, which can be orders of magnitude slower than register access (the “von Neumann Bottleneck”)
- Fast registers are (today) a scarce resource, shared by many code fragments
- How can registers be used most effectively?
  - Requires a careful examination of a program’s structure
  - Analysis phase: building data structures (typically directed graphs) that capture definition/use relationships
  - Transformation phase: using this information to rewrite code, attempting to most efficiently utilise registers
- Problem is NP-complete
- One of the central topics of Part II Optimising Compilers.
- Here we focus only on general issues: calling conventions and register spilling.
Caller/callee conventions

• Caller and callee code may use overlapping sets of registers
• An agreement is needed concerning use of registers
  • Are some arguments passed in specific registers?
  • Is the result returned in a specific register?
  • If the caller and callee are both using a set of registers for “scratch space” then caller or callee must save and restore these registers so that the caller’s registers are not obliterated by the callee.
• Standard calling conventions identify specific subsets of registers as “caller saved” or “callee saved”
  • **Caller saved**: if caller cares about the value in a register, then must save it before making any call
  • **Callee saved**: The caller can be assured that the callee will leave the register intact (perhaps by saving and restoring it)
Another C example.
X86, 64 bit, with gcc

```c
int callee(int, int, int, int, int, int, int);

int caller(void)
{
    int ret;
    ret = callee(1, 2, 3, 4, 5, 6, 7);
    ret += 5;
    return ret;
}
```

```assembly
_caller:
pushq  %rbp              # save frame pointer
movq   %rsp, %rbp        # set new frame pointer
subq   $16, %rsp         # make room on stack
movl   $7, (%rsp)        # put 7th arg on stack
movl   $1, %edi          # put 1st arg on in edi
movl   $2, %esi          # put 2nd arg on in esi
movl   $3, %edx          # put 3rd arg on in edx
movl   $4, %ecx          # put 4th arg on in ecx
movl   $5, %r8d          # put 5th arg on in r8d
movl   $6, %r9d          # put 6th arg on in r9d
callq  _callee           #will put result in eax
addl   $5, %eax          # add 5
addq   $16, %rsp         # adjust stack
popq   %rbp              # restore frame pointer
ret     # pop return address, go there
```
Register spilling

- What happens when all registers are in use?
- Could use the stack for scratch space …
- … or (1) move some register values to the stack, (2) use the registers for computation, (3) restore the registers to their original value
- This is called register spilling.
Simple optimisations.  
Inline expansion

```
fun f(x) = x + 1
fun g(x) = x - 1
...
...
fun h(x) = f(x) + g(x)
```

inline f and g

```
fun f(x) = x + 1
fun g(x) = x - 1
...
...
fun h(x) = (x+1) + (x-1)
```

(+): Avoid building activation records at runtime
(+): May allow further optimisations

(-): May lead to “code bloat”
    (apply only to functions with “small” bodies?)

Question: if we inline all occurrences of a function, can we delete its definition from the code?
What if it is needed at link time?
Be careful with variable scope

Inline g in h

What kind of care might be needed will depend on the representation level of the Intermediate code involved.

NO

let val x = 1
  fun g(y) = x + y
  fun h(x) = g(x) + 1
in
  h(17)
end

YES

let val x = 1
  fun g(y) = x + y
  fun h(x) = x + y + 1
in
  h(17)
end

let val x = 1
  fun g(y) = x + y
  fun h(z) = x + z + 1
in
  h(17)
end
Propagate constants and evaluate simple expressions at compile-time.

Note: opportunities are often exposed by inline expansion!

David Gries:
“Never put off till run-time what you can do at compile-time.”

But be careful

How about this?
Replace
\[ x \ast 0 \]
with
\[ 0 \]

OOPS, not if \( x \) has type float!

\[ \text{NAN} \ast 0 = \text{NAN}, \]
Peephole Optimization

W. M. McKeeman
Stanford University, Stanford, California

Communications of the ACM, July 1965

Example 1. Source code:

\[ X := Y; \]
\[ Z := X + Z \]

Compiled code:

- LDA Y load the accumulator from Y
- STA X store the accumulator in X
- LDA X load the accumulator from X
- ADD Z add the contents of Z
- STA Z store the accumulator in Z

Eliminate!

Results for syntax-directed code generation.
peephole optimisation

Sweep a window over the code sequence looking for instances of simple code patterns that can be rewritten to better code ...

(might be combined with constant folding, etc, and employ multiple passes)

Examples
-- eliminate useless combinations (push 0; pop)
-- introduce machine-specific instructions
-- improve control flow. For example: rewrite
  “GOTO L1 ... L1: GOTO L2”
to
  “GOTO L2 ... L1 : GOTO L2”
gcc example.

-O<m> turns on optimisation to level m

```c
int h(int n) { return (0 < n) ? n : 101; }
```

```c
int g(int n) { return 12 * h(n + 17); }
```

```
g.c
 gcc –O2 –S –c g.c

_g:
  .cfi_startproc
  pushq  %rbp
  movq  %rsp, %rbp
  addl  $17, %edi
  imull  $12, %edi, %ecx
  testl  %edi, %edi
  movl  $1212, %eax
  cmovgl  %ecx, %eax
  popq  %rbp
  ret
  .cfi_endproc
```

Wait. What happened to the call to h???

GNU AS (GAS) Syntax

x86, 64 bit
gcc example (-O<m> turns on optimisation)

```
g.c

int h(int n)  { return (0 < n) ? n : 101 ; }
int g(int n)  { return 12 * h(n + 17); }
```

The compiler must have done something similar to this:

```
int g(int n)  { return 12 * h(n + 17); }

⇒
int g(int n)  { int t := n+ 17; return 12 * h(t); }

⇒
int g(int n)  { int t := n+ 17; return 12 *((0 < t) ? t : 101 ); }

⇒
int g(int n)  { int t := n+ 17; return (0 < t) ? 12 * t : 1212 ; }

⇒ ...
```
New Topic: OOP Objects (single inheritance)

let start := 10

class Vehicle extends Object {
    var position := start
    method move(int x) = {position := position + x}
}
class Car extends Vehicle {
    var passengers := 0
    method await(v : Vehicle) =
        if (v.position < position)
            then v.move(position - v.position)
        else self.move(10)
}
class Truck extends Vehicle {
    method move(int x) =
        if x <= 55 then position := position + x
}
var t := new Truck
var c := new Car
var v : Vehicle := c
in
    c.passengers := 2;
    c.move(60);
    v.move(70);
    c.await(t)
end

method override

subtyping allows a Truck or Car to be viewed and used as a Vehicle
Object Implementation?

- how do we access object fields?
  - both inherited fields and fields for the current object?
- how do we access method code?
  - if the current class does not define a particular method, where do we go to get the inherited method code?
  - how do we handle method override?
- How do we implement subtyping ("object polymorphism")?
  - If B is derived from A, then need to be able to treat a pointer to a B-object as if it were an A-object.
Another OO Feature

• Protection mechanisms
  – to encapsulate local state within an object, Java has “private” “protected” and “public” qualifiers
    • private methods/fields can’t be called/used outside of the class in which they are defined
  – This is really a scope/visibility issue! Front-end during semantic analysis (type checking and so on), the compiler maintains this information in the symbol table for each class and enforces visibility rules.
Object representation

```cpp
class A {
public:
    int a1, a2;
    virtual void m1(int i) {
        a1 = i;
    }
    virtual void m2(int i) {
        a2 = a1 + i;
    }
}
```

NB: a compiler typically generates methods with an extra argument representing the object (self) and used to access object data.
Inheritance ("pointer polymorphism")

```cpp
class B : public A {
public:
    int b1;

virtual void m3(void) {
    b1 = a1 + a2;
}
}
```

Note that a pointer to a B object can be treated as if it were a pointer to an A object!
class C : public A {
    public:
        int c1;

        virtual void m3(void) {
            b1 = a1 + a2;
        }
        virtual void m2(int i) {
            a2 = c1 + i;
        }
    }
}

Method overriding

Object data

Vtable for class C

Declared

Defined
Static vs. Dynamic

• which method to invoke on overloaded polymorphic types?

```cpp
class C *c = ...;
class A *a = c;
a->m2(3);
```

???

```
m2_A_A(a, 3);
m2_A_C(a, 3);
```

static
dynamic
Dynamic dispatch implemented with vtables

A pointer to a class C object can be treated as a pointer to a class A object.

```
class C *c = ...;
class A *a = c;
a->m2(3);
*a->vtable[1])(a, 3);
```
If expression e evaluates “normally” to value v, then v is the result of the entire expression.

Otherwise, an exceptional value v’ is “raised” in the evaluation of e, then result is (f v’)

Evaluate expression e to value v, and then raise v as an exceptional value, which can only be “handled”.

Implementation of exceptions may require a lot of language-specific consideration and care. Exceptions can interact in powerful and unexpected ways with other language features. Think of C++ and class destructors, for example.
Viewed from the call stack

Call stack just before evaluating code for

e handle f

Push a special frame for the handle

“raise v” is encountered while evaluating a function body associated with top-most frame

“Unwind” call stack. Depending on language, this may involve some “clean up” to free resources.
Possible pseudo-code implementation

```
let fun _h27 () =
  build special "handle frame"
  save address of f in frame;
  ... code for e ...
  return value of e
in _h27 () end

raise e

... code for e ...
save v, the value of e;
unwind stack until first
fp found pointing at a handle frame;
Replace handle frame with frame
for call to (extracted) f using
v as argument.
```
Lecture 15
Automating run-time memory management

• Managing the heap
• Garbage collection
  – Reference counting
  – Mark and sweep
  – Copy collection
  – Generational collection

Read Chapter 12 of Basics of Compiler Design (T. Mogensen)
Explicit (manual) memory management

- User library manages memory; programmer decides when and where to allocate and de-allocate
  - void* malloc(long n)
  - void free(void *addr)
  - Library calls OS for more pages when necessary
  - **Advantage**: Gives programmer a lot of control.
  - **Disadvantage**: people too clever and make mistakes. Getting it right can be costly. And don’t we want to automate-away tedium?
  - **Advantage**: With these procedures we can implement memory management for “higher level” languages ;-)
Memory Management

- Many programming languages allow programmers to (implicitly) allocate new storage dynamically, with no need to worry about reclaiming space no longer used.
  - New records, arrays, tuples, objects, closures, etc.
  - Java, SML, OCaml, Python, JavaScript, Python, Ruby, Go, Swift, SmallTalk, …

- Memory could easily be exhausted without some method of reclaiming and recycling the storage that will no longer be used.
  - Often called “garbage collection”
  - Is really “automated memory management” since it deals with allocation, de-allocation, compaction, and memory-related interactions with the OS.
Automation is based on an approximation: if data can be reached from a root set, then it is not “garbage.”

Type information required (pointer or not), some kind of “tagging” needed.
Identify Cells Reachable From Root Set...
... reclaim unreachable cells
But How? Two basic techniques, and many variations

- **Reference counting**: Keep a reference count with each object that represents the number of pointers to it. Is garbage when count is 0.
- **Tracing**: find all objects reachable from root set. Basically transitive close of pointer graph.

For a very interesting (non-examinable) treatment of this subject see

**A Unified Theory of Garbage Collection.**

David F. Bacon, Perry Cheng, V.T. Rajan.

OOPSLA 2004.

In that paper reference counting and tracing are presented as “dual” approaches, and other techniques are hybrids of the two.
Reference Counting, basic idea:

- Keep track of the number of pointers to each object (the reference count).
- When Object is created, set count to 1.
- Every time a new pointer to the object is created, increment the count.
- Every time an existing pointer to an object is destroyed, decrement the count.
- When the reference count goes to 0, the object is unreachable garbage.
Reference counting can’t detect cycles!

Cons
- Space/time overhead to maintain count.
- Memory leakage when have cycles in data.

Pros
- Incremental (no long pauses to collect…)
Mark and Sweep

• A two-phase algorithm
  – Mark phase: Depth first traversal of object graph from the roots to mark live data
  – Sweep phase: iterate over entire heap, adding the unmarked data back onto the free list
• Basic idea: use 2 heaps
  – One used by program
  – The other unused until GC time

• GC:
  – Start at the roots & traverse the reachable data
  – Copy reachable data from the active heap (from-space) to the other heap (to-space)
  – Dead objects are left behind in from space
  – Heaps switch roles
Copying Collection

```
from-space

roots

```

```
to-space

```
Copying GC

• **Pros**
  – Simple & collects cycles
  – Run-time proportional to # live objects
  – Automatic compaction eliminates fragmentation

• **Cons**
  – Twice as much memory used as program requires
    • Usually, we anticipate live data will only be a small fragment of store
    • Allocate until 70% full
    • From-space = 70% heap; to-space = 30%
  – Long GC pauses = bad for interactive, real-time apps
OBSERVATION: for a copying garbage collector

- 80% to 98% new objects die very quickly.
- An object that has survived several collections has a bigger chance to become a long-lived one.
- It’s inefficient that long-lived objects be copied over and over.

Diagram from Andrew Appel’s *Modern Compiler Implementation*
IDEA: Generational garbage collection

Segregate objects into multiple areas by age, and collect areas containing older objects less often than the younger ones.

Diagram from Andrew Appel’s Modern Compiler Implementation
Other issues...

- When do we promote objects from young generation to old generation
  - Usually after an object survives a collection, it will be promoted
- Need to keep track of older objects pointing to newer ones!
- How big should the generations be?
  - When do we collect the old generation?
  - After several minor collections, we do a major collection
- Sometimes different GC algorithms are used for the new and older generations.
  - Why? Because the have different characteristics
  - Copying collection for the new
    - Less than 10% of the new data is usually live
    - Copying collection cost is proportional to the live data
  - Mark-sweep for the old
LECTURE 16
Bootstrapping a compiler

• Compilers compiling themselves!
• Read Chapter 13 Of
  • Basics of Compiler Design
  • by Torben Mogensen
    http://www.diku.dk/hjemmesider/ansatte/torbenm/Basics/

http://mythologian.net/ouroboros-symbol-of-infinity/
Bootstrapping. We need some notation . . .

An application called **app** written in language **A**

An interpreter or VM for language **A**
Written in language **B**

A machine called **mch** running language **A** natively.

**Simple Examples**

```
hello
x86
x86
M1
```

```
hello
JBC
JBC
jvm
x86
x86
M1
```
This is an application called *trans* that translates programs in language **A** into programs in language **B**, and it is written in language **C**.
Ahead-of-time compilation

Thanks to David Greaves for the example.
Of course translators can be translated

Translator \texttt{foo.B} is produced as output from \texttt{trans} when given \texttt{foo.A} as input.
Our seemingly impossible task

We have just invented a really great new language \textbf{L} (in fact we claim that “\textbf{L} is far superior to C++”). To prove how great \textbf{L} is we write a compiler for \textbf{L} in \textbf{L} (of course!). This compiler produces machine code \textbf{B} for a widely used instruction set (say \textbf{B} = x86).

Furthermore, we want to compile our compiler so that it can run on a machine running \textbf{B}.

Our compiler is written in \textbf{L}!

How can we compiler our compiler?

There are many many ways we could go about this task. The following slides simply sketch out one plausible route to fame and fortune.
Step 1
Write a small interpreter (VM) for a small language of byte codes

**MBC** = My Byte Codes

The **zoom** machine!
Step 2
Pick a small subset $S$ of $L$ and write a translator from $S$ to MBC

Write `comp_1.cpp` by hand. (It sure would be nice if we could hide the fact that this is written in C++.)

Compiler `comp_1.B` is produced as output from `gcc` when `comp_1.cpp` is given as input.
Step 3
Write a compiler for $L$ in $S$

Write a compiler $\text{comp}_2.S$ for the full language $L$, but written only in the sub-language $S$.

Compile $\text{comp}_2.S$ using $\text{comp}_1.B$ to produce $\text{comp}_2.mbc$
Step 4
Write a compiler for L in L, and then compile it!

Rewrite/extend compiler comp_2.S to produce comp.L using the full power of language L.

We have achieved our goal!
We wrote these compilers and the MBC VM.
Step 5: Cover our tracks and leave the world mystified and amazed!

Our **L** compiler download site contains only three components:

1. Use `gcc` to compile the **zoom** interpreter
2. Use **zoom** to run **mr-e** with input **comp.L** to output the compiler **comp.B**. MAGIC!

**comp_2.mbc** is a just file of **bytes**. We give it the mysterious name such as **mr-e**

Shhhh! Don’t tell anyone that we wrote the first compiler in C++
Solving a different problem.

You have:
1. An ML compiler on ARM. Who knows where it came from.
2. An ML compiler written in ML, generating x86 code.

You want:
An ML compiler generating x86 and running on an x86 platform.