

Compiler Construction

University of Cambridge

Ashwin Ahuja

Computer Science Tripos Part IB
Paper 4

April 2019

Contents

1	Structure	3
1.1	Front End	3
1.2	Middle	4
1.3	Back End	4
1.4	SLANG	4
1.4.1	Syntax	4
1.4.2	Front End for SLANG	4
1.4.3	Parsed AST	4
1.4.4	Internal AST	5
2	Lexing and Parsing	6
2.1	Theory of Regular Languages and Finite Automata	6
2.1.1	Constructing a lexer	7
2.2	Context-Free Grammars	7
2.2.1	Derivations	8
2.2.2	Language	8
2.3	Ambiguity Problem	8
2.4	Generating a Lexical Analyser	9
2.5	Recursive Descent Parsing	9
2.5.1	Eliminating Left Recursion	9
2.5.2	Required Information	9
2.6	LL and LR	10
2.6.1	NFA for LR(0) items	12
2.6.2	Building DFA States	12
2.6.3	Parsing with LR Table	13
2.6.4	LR(0)	13
2.6.5	LR(1)	13
2.7	Auto-Generation Tools - LEX, YACC	14
3	Interpreters	14
3.1	Definitional Interpreter - Interpreter 0	14
3.1.1	Code	15
3.2	Interpreter 2	16
3.3	Interpreter 3	18
3.3.1	Code	19
3.4	Jargon Virtual Machine	21
3.4.1	Structure	21
3.4.2	Item Code	22
3.4.3	Finding variable's value at runtime	22
3.4.4	How operations work?	24
3.4.5	Notes	26
4	Recursion to Iteration Transforms	26
4.1	Tail Recursion Elimination	26
4.2	Continuation Passing Style (CPS)	26
4.2.1	cnt	27
4.2.2	Identity Continuation	27
4.3	Defunctionalisation (DFC)	27
4.4	Evaluating Expressions Example	28
5	Compilers in OS context	30
5.1	Portability	30
5.2	Linker	31
5.3	Application Binary Interface	31
5.4	Runtime System	32
5.5	Memory Layout in UNIX	33
5.6	x86 Assembler	33
5.6.1	Jargon VM instructions in X86	33
5.6.2	Arithmetic	34

6	Performance improvements	34
6.1	Stacks vs Registers	34
6.1.1	Caller / Callee Conventions	34
6.1.2	Register Spilling	34
6.2	Inline Expansion	35
6.3	Constant Propagation - Constant Folding	35
6.4	Peephole Optimisation	35
7	OOP object representations	35
8	Exceptions on stack	35
9	Runtime Memory Management	36
9.1	Explicit Memory Management	36
9.2	Automated Memory Management	36
10	Bootstrapping a Compiler	37
10.1	Notation	37
10.2	Instructions for use	38

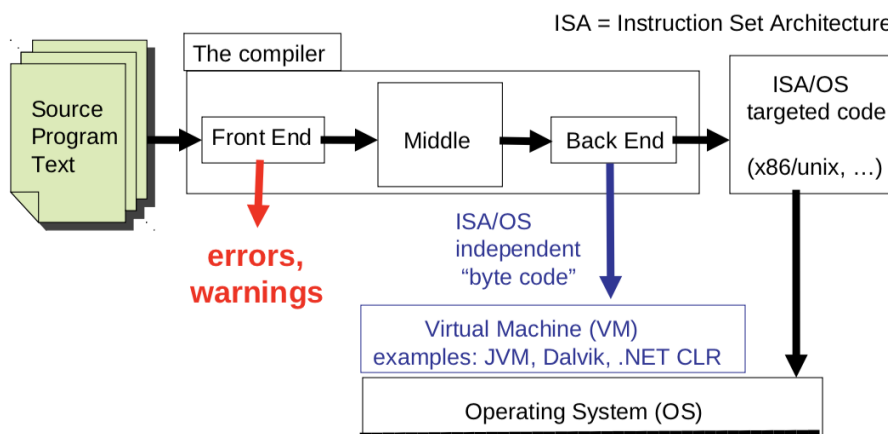
1 Structure

Compiler effectively takes a source program text into a program for a target machine.

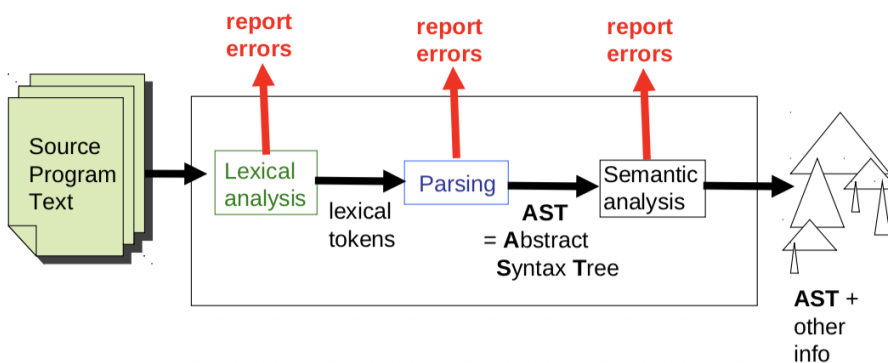
Goals:

1. Meaning is preserved - therefore correct
2. Produce usable error messages
3. Generate efficient code
4. Itself be efficient
5. Be well-structured and maintainable

It bridges the gap between high level languages which are: (1) machine independent, (2) complex syntax, (3) Complex type system, (4) Variables, (5) Nested scope, (6) Procedures, functions, (7) Objects, (8) Modules and typical target languages which are: (1) Machine specific, (2) Simple syntax, (3) Simple types, (4) Memory, registers, words, (5) Single flat scope



1.1 Front End



Lexical Analysis: Based on finite automaton and regular expression

Parsing: Based on push-down automaton and context free grammars

Semantic Analysis: Enforces the static semantics of the language including type checking, functions, etc

1.2 Middle

Takes the AST and other info and converts into low-level, while carrying out the optimisations as required. Output is a low-level retargetable representation.

Our view of the system suggests that there are a sequence of small transformations between a set of intermediate languages where:

1. Each intermediate language has its own semantics (may be informal)

2. Each transformation preserved semantics
3. Each transformation eliminates only a few aspects of the gap
4. Each transformation is fairly easy to understand
5. Some transformations can be described as optimizations
6. Associate each intermediate language with its own interpreter

1.3 Back End

Takes the low-level retargetable representation and converts it to the specific architecture or type that would allow the user to run the code - requires knowledge of instruction set and details of target machine. In this place, you can also have target-dependent optimisations.

1.4 SLANG

Simple Language - subset of L3 with ugly concrete semantics

1.4.1 Syntax

```

uop ::= - | ~(where ~is boolean negation)
bop ::= + | - | * | < | = | && | ||
t ::= bool | int | unit | (t) | t * t | t + t | t -> t | t ref
e ::= () | n | true | false | x | (e) | ? - request integer input from terminal | e bop e | uop e | if e then e else e
end | e e | fun (x : t) -> e end | let x : t = e in e end | let f(x : t) : t = e in e end | !e | ref e | e := e | while e do
e end | begin e; e; ... e end | (e, e) | snd e | fst e | inl t e | inr t e | case e of inl(x : t) -> e | inr(x : t) -> e end

```

1.4.2 Front End for SLANG

1. **Parsing** - takes input file and converts to a parsed AST
2. **Static Analysis** - takes parsed AST and changes it and outputs a parsed AST. In this step, check types and context-sensitive rules and resolve overloaded operators
3. Then, removes the syntactic sugar, including file location information and most type information - creating an intermediate AST

1.4.3 Parsed AST

```

type var = string
type loc = Lexing.position #used to generate error messages
type type_expr = TEint
  | TEbool
  | TEunit
  | Teref of type_expr
  | TEarrow of type_expr * type_expr
  | TEproduct of type_expr * type_expr
  | TEunion of type_expr * type_expr

type oper = ADD | MUL | SUB | LT | AND | OR | EQ | EQB | EQ
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 * type_expr * expr
| LetRecFun of loc * var * lambda * type_expr * expr

```

static.ml

1. Check type correctness
2. Rewrite expressions to resolve EQ to EQI (for integers) or EQB (for booleans)
3. Only LetFun is returned by parser and rewrite to LetRecFun when the function is actually recursive

```

val infer : (Past.var * Past.type_expr) list -> (Past.expr * Past.type_expr)
val check : Past.expr -> Past.expr # infer on an empty environment

```

1.4.4 Internal AST

These have no locations and types and no Let and EQ.

```

type var = string
type oper = ADD | MUL | SUB | LT | AND | OR | EQB | EQI
type unary_oper = NEG | NOT | READ

```

```

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

```

and lambda = var * expr

In `past_to_ast.ml`:

```
val translate_expr : Past.expr -> Ast.expr
```

Example: `'let x : t = e1 in e2 end' -> '(fun (x : t) -> e2 end) e1'`

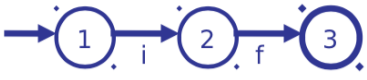
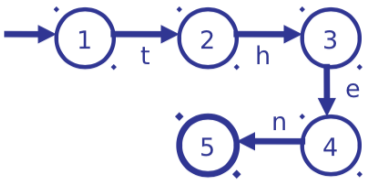
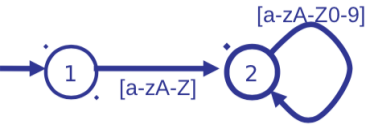
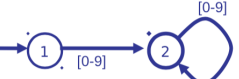
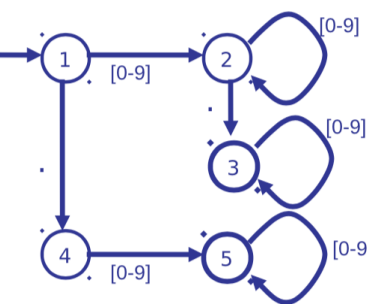
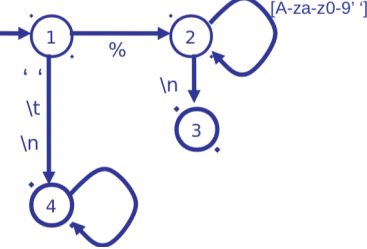
2 Lexing and Parsing

Problem: (1) Translate a sequence of characters into (2) a sequence of tokens (3) implemented with some data type

2.1 Theory of Regular Languages and Finite Automata

Lexing Problem

- Given an ordered list of regular expressions: e_1, e_2, \dots, e_k and an input string w
- Find pairs $(i_1, w_1), (i_2, w_2), \dots, (i_n, w_n)$ st
 - $w = w_1 w_2 \dots w_n$
 - $w_j \in L(e_{i_j})$
 - Priority Rule:** $w_j \in L(e_s) \rightarrow i_j \leq s$
 - Longest Match:** $\forall j \forall u \in \text{prefix}(w_{j+1} w_{j+2} \dots w_n) : u \neq \epsilon \rightarrow \forall s w_j u \notin L(e_s)$

Regular Expression	Finite Automata	Token
Keyword: if		KEY(IF)
Keyword: then		KEY(then)
Identifier: [a-zA-Z][a-zA-Z0-9]*		ID(s)
number: [0-9][0-9]*		NUM(n)
real: ([0-9] + '.' [0-9]*) ([0-9]* '.' [0-9]) +		NUM(n)
White-space: (' ' '\n' '\t') + '%' [A-Za-z0-9' '] + '\n'		No token

Parse Tree = derivation tree = concrete syntax - contains all the information (in not necessarily the most efficient storage systems)

Abstract Syntax Tree: contains **only** the information needed to generate an intermediate representation - generally compiler only implicitly constructs a concrete syntax tree in the parsing process and explicitly creates

an AST

2.1.1 Constructing a lexer

1. INPUT: an ordered list of regular expressions
2. Construct all corresponding finite automata
3. Construct a single non-deterministic finite automata
4. Construct a single deterministic finite automata
5. **Using Longest Match**

This is used in order to work out which path to go down - you do the following (1) Start in initial state then **repeat** (2) read input until dead state is reached. Emit token associated with last accepting state, (3) reset state to start state

2.2 Context-Free Grammars

$E ::= ID \mid NUM \mid E * E \mid E / E \mid E + E \mid E - E \mid (E)$

where:

- E is a non-terminal symbol
- ID and NUM are lexical symbols
- $E ::= \dots$ are production rules
- $*, (,), +, -$ are terminal symbols

Context Free Grammar 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 are disjoint)
- $S \in N$ is the start symbol
- $R \subseteq N \times (N \cup T)^*$ is a set of rules

2.2.1 Derivations

1. Start from start symbol S
2. Productions used to derive a sequence of tokens from the start symbol
3. For arbitrary string a, b and c comprised of both terminal and non terminal symbols and a production $A \rightarrow b$, a single step of derivation is $aAc \Rightarrow abc$
4. $\forall a \Rightarrow^* b$ means that b can be derived from a in 0 or more single steps
5. $\forall a \Rightarrow^+ b$ means that b can be derived from a in 1 or more single steps

2.2.2 Language

$L(G)$ is the language generated by grammar G - set of all terminal symbols derived from the start symbol S:

$$L(G) = \{w \in T^* \mid S \Rightarrow^+ w\} \quad (1)$$

That is, 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 over T

2.3 Ambiguity Problem

Issues:

1. CFGs describe how to generate - parsing is the inverse of this...
2. Given an input string, is it in the language generated by a CFG - then construct a derivation tree.
However, the derivation trees are not unique - can have multiple derivation trees which correspond to the same strings

Can try to modify grammar in order to eliminate ambiguity hence G2 and G4

G2

```
S ::= E
E ::= E + T | E - T | T
T ::= T * F | T / F | F
F ::= NUM | ID | (E)
```

G4

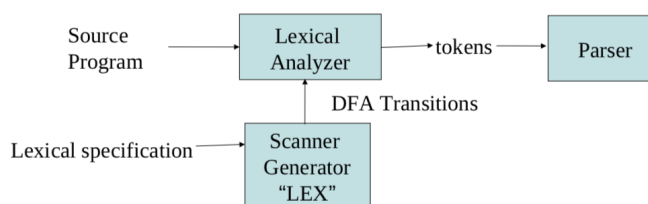
```
S ::= WE | NE
WE ::= if E then WE else WE | ...
NE ::= if E then S | if E then WE else NE
```

Important points

1. Some context free languages are **inherently ambiguous** - \nexists context-free grammar where it is not ambiguous
2. Checking for ambiguity in an arbitrary context-free grammar is an undecidable problem
3. Given two grammars G1 and G2, checking $L(G1) = L(G2)$ is not decidable

2.4 Generating a Lexical Analyser

Use regular expressions as basis of a lexical specification then the core of the lexical analyzer is just a deterministic finite automata



2.5 Recursive Descent Parsing

Top-down method of syntax analysis in which a set of recursive procedures are used to process the input. One procedure is associated with each non-terminal of a grammar.

Relies upon information about the first symbols that can be generated by a production body. Therefore need the first symbol for each production rule to be different! Eliminating left recursion fixes this

However, issue is that it is possible for recursive-descent parsers to loop forever - especially where there is left-recursive productions: $\text{expr} \rightarrow \text{expr} + \text{term}$

2.5.1 Eliminating Left Recursion

Say $A ::= A\alpha|\beta$ where α and β are sequences of terminals and non-terminals that do not start with A. Convert this to:

1. $A' ::= \alpha A' \mid \epsilon$
2. $A ::= \beta A'$

Therefore, G2 becomes G6:

$S ::= E$

$E ::= TE'$

$E' ::= + TE' \mid -TE' \mid \epsilon$

$T ::= FT'$

$T' ::= * FT' \mid / FT' \mid \epsilon$

$F ::= \text{NUM} \mid \text{ID} \mid (E)$

2.5.2 Required Information

1. **FIRST[X]**: set of terminal symbols that can begin strings derived from X
2. **FOLLOW[X]**: set of terminal symbols that can immediately follow X in some derivation
3. **nullable[X]**: true if X can derive the empty string, false otherwise

Calculating 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 until no change
 - For each production $X \rightarrow Y_1 Y_2 \dots Y_k$:
 - * If Y_1, \dots, Y_k are all nullable, or $k = 0$, then $\text{nullable}[X] := \text{true}$
 - * foreach i from 1 to k, each j from i+1 to k:
 - if $Y_1 \dots Y(i-1)$ are all nullable or $i = 1$ then
 $\text{FIRST}[X] := \text{FIRST}[X] \cup \text{FIRST}[Y(i)]$
 - if $Y(i+1) \dots Y_k$ are all nullable or if $i = k$ then
 $\text{FOLLOW}[Y(i)] := \text{FOLLOW}[Y(i)] \cup \text{FOLLOW}[X]$
 - if $Y(i+1) \dots Y(j-1)$ are all nullable or $i+1 = j$ then
 $\text{FOLLOW}[Y(i)] := \text{FOLLOW}[Y(i)] \cup \text{FIRST}[Y(j)]$

G6

	Nullable	FIRST	FOLLOW
S	False	{ (, ID, NUM }	{ }
E	False	{ (, ID, NUM }	{), \$ }
E'	True	{ +, - }	{), \$ }
T	False	{ (, ID, NUM }	{), +, -, \$ }
T'	True	{ *, / }	{), +, -, \$ }
F	False	{ (, ID, NUM }	{), *, /, +, -, \$ }

Use this information to create the Predictive Parsing Table which states which rule we could be in given a particular symbol, or more formally: $\text{Table}[X, T] = \text{Set of productions where:}$

$X ::= Y_1 \dots Y_k$ in $\text{Table}[X, T]$ if $T \in \text{FIRST}[Y_1 \dots Y_k]$ or if $(T \in \text{FOLLOW}[X] \text{ AND } \text{nullable}[Y_1 \dots Y_k])$

	+	*	()	ID	NUM	\$
S			$S ::= E\$$		$S ::= E\$$	$S ::= E\$$	
E			$E ::= TE'$		$E ::= TE'$	$E ::= TE'$	
E'	$E' ::= +TE'$			$E' ::=$			$E' ::=$
T			$T ::= FT'$		$T ::= FT'$	$T ::= FT'$	
T'	$T' ::=$	$T' ::= *FT'$		$T' ::=$			$T' ::=$
F			$F ::= (E)$		$F ::= ID$	$F ::= NUM$	

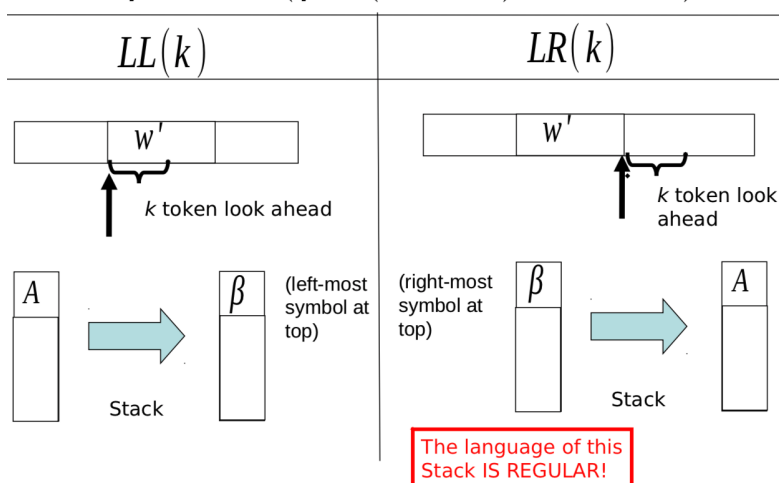
If there are multiple entries in a single cell, you can't do recursive descent parsing

2.6 LL and LR

- **LL(k)**: Left-to-right parse, Left-most derivation, k-symbol lookahead. Therefore, by looking at the next k tokens an LL(k) parser predicts the next production - above was an example of LL(1)
- **LR(k)**: Left-to-right parse, Right-most-derivation, k-symbol lookahead. Therefore postpones the production selection until the right-hand-side has been seen and as many as k symbols beyond
- **LALR(1)**: subclass of LR(1)

LL(k) vs LR(k) reductions

$$A \rightarrow \beta \Rightarrow w' \quad (\beta \in (T \cup N), w' \in T)$$



In an LL parser, there are the two actions:

1. **Predict**: Based on the leftmost terminal and some number of lookahead tokens, choose which production to apply to get closer to input string
2. **Match**: Match leftmost terminal symbol with leftmost unconsumed symbol of input

whereas in LR, the actions are:

1. **Shift**: Add the next token of input to a buffer for consideration
2. **Reduce**: Reduce a collection of terminals and non-terminals in this buffer back to a non-terminal by reversing a production

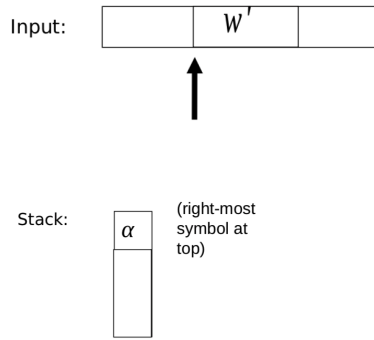
When to shift / reduce: Build an FSA from LR(0) items - indicate what is on the stack (to the left of \cdot) and what is still in the input stream. For example if: (1) $S ::= A$ and $A ::= (A) \mid ()$ then the items are:

$S ::= \cdot A$
 $S ::= A \cdot$
 $A ::= \cdot (A)$
 $A ::= (\cdot A)$
 $A ::= (A \cdot)$
 $A ::= (A) \cdot$
 $A ::= \cdot ()$
 $A ::= (\cdot)$
 $A ::= () \cdot$

LR Items

- If current state contains 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 with the next state containing $A ::= \alpha c \cdot \beta$
- If the state contains the item $A ::= \alpha \cdot$ - the state prompts the parser to perform a reduce action
- If the state contains the item $S ::= \alpha \cdot$ and input buffer is empty then the state prompts the parser to accept

LR(k) States - non-deterministic State: $A \rightarrow \alpha \cdot \beta, a_1 a_2 \dots a_k$ represents the situation:



With $\beta a_1 a_2 \dots a_k \Rightarrow w'$

2.6.1 NFA for LR(0) items

Transitions 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. Exists an ϵ -transition from item $A ::= \alpha \cdot X\beta$ to $X ::= \cdot y$ where X is a non-terminal
4. $S ::= \cdot A$ is the start state
5. $A ::= \alpha \cdot$ is a final state

DFA for LR(0) parsing can be obtained by the usual NFA to DFA construction, requiring: (1) ϵ -closure, (2) $\text{move}(S, a)$

Fixed Point Algorithm

1. Every item in I is also an item in $\text{Closure}(I)$
2. If $A ::= \alpha \cdot B\beta$ is in $\text{Closure}(I)$ and $B ::= \cdot y$ is an item, then add $B ::= \cdot y$ to $\text{Closure}(I)$
3. Repeat until no more new items can be added to $\text{Closure}(I)$

Goto() $\text{Goto}()$ of a set of items finds the new state after consuming a grammar symbol while in the current state, that is:

$$\text{Goto}(I, X) = \text{Closure}(\{A ::= \alpha X \cdot \beta \mid A ::= \alpha \cdot X\beta \text{ in } I\}) \quad (2)$$

2.6.2 Building DFA States

- Let A be start symbol and S a new start symbol: $S ::= A$
- First state is $\text{Closure}(\{S ::= \cdot A\})$
- For 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 are possible

DFA states can be converted into a parse table with symbols and whether it includes a shift or a reduce.

2.6.3 Parsing with LR Table

Use table and the top-of-stack and input symbol to get output action (stored in action table), which does the following:

1. **shift sn**: (1) advance input one token, (2) push sn on stack
2. **reduce $X ::= \alpha$** :
 - (a) Pop stack $2*|\alpha|$ times (grammar symbols are paired with states). In the state now on top of stack, use goto table to get next state sn
 - (b) Push it on top of stack
3. **accept**: stop and accept
4. **error**: produce error message

Example

PARSE STACK	REMAINING INPUT	PARSER ACTION
	(id + id)\$	Shift (push next token from input on stack, advance input)
(id + id)\$	Shift
(id	+ id)\$	Reduce: $T \rightarrow id$ (pop right-hand side of production off stack, push left-hand side, no change in input)
(T	+ id)\$	Reduce: $E \rightarrow T$
(E	+ id)\$	Shift
(E +	id)\$	Shift
(E + id)\$	Reduce: $T \rightarrow id$
(E + T)\$	Reduce: $E \rightarrow E + T$ (Ignore: $E \rightarrow T$)
(E)\$	Shift
(E)	\$	Reduce: $T \rightarrow (E)$
T	\$	Reduce: $E \rightarrow T$
E	\$	Reduce: $S \rightarrow E$
S	\$	

2.6.4 LR(0)

1. No lookahead
2. Vulnerable to unnecessary conflicts: (1) Shift/reduce conflicts (may reduce too soon), (2) Reduce/reduce conflicts

This is fixed using LR(1) adding in systematic lookahead

2.6.5 LR(1)

LR(1) item: is a pair - $(X ::= \alpha \cdot \beta, a)$ where:

- $X ::= \alpha \cdot \beta$ is a production rule
- a is a terminal (lookahead terminal) - LR(1) means a single lookahead terminal

It describes a context of the parser - attempting to find an X followed by an a , and already have an α on top of the stack. Therefore, need to see next a prefix derived from βa

Closure

$\text{Closure}(\text{items}) =$

repeat:

```
    foreach  $[X ::= \alpha \cdot Y\beta, a]$  in Items
        foreach production  $Y ::= y$ 
            foreach  $b$  in  $\text{First}(\beta a)$ 
                add  $[Y ::= \cdot y, b]$  to Items
```

until Items is unchanged

Constructing Parsing DFA

- DFA state is closed set of LR(1) items
- Start state is $(S' ::= \cdot S\$, \text{dummy})$
- State that contains $[X ::= \alpha \cdot, b]$ is labelled with 'reduce with $X ::= \alpha$ on lookahead b
- State s that contains $[X ::= \alpha \cdot Y\beta, b]$ has a transition labelled y to the state obtained from $\text{Transition}(s, Y)$ where Y can be terminal or non-terminal

```
Transition(s, Y):
    Items = {}
    foreach  $[X ::= \alpha \cdot Y\beta, b]$  in s
        add  $[X ! \alpha Y \cdot \beta, b]$  to Items
    return Closure(Items)
```

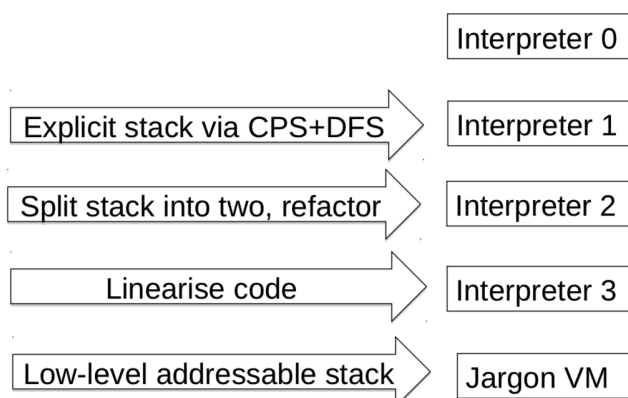
Parse Table

- Shift and goto as before
- **Reduce:** state I with item $(A \rightarrow \alpha \cdot, z)$ gives a reduce $A \rightarrow \alpha$ if z is the next character in the input

2.7 Auto-Generation Tools - LEX, YACC

1. Slow
2. Hard to generate good error messages for compiler users
3. Often need to tweak grammar, but tool messages can be obscure

3 Interpreters



3.1 Definitional Interpreter - Interpreter 0

- Slang / L3 values represented directly as OCaml values
- Recursive interpreter implements a **denotational semantics**
 - N = set of integers, B = set of booleans, A = set of addresses, I = set of identifiers, Expr = set of L3 expressions
 - E = set of environments = $I \rightarrow V$
 - S = set of stores = $A \rightarrow V$
 - V = set of value $\approx A$
 - + B
 - + N
 - + $\{()\}$
 - + $V \times V$
 - + $(V + V)$
 - + $(V \times S) \rightarrow (V \times S)$
 - M = meaning function : $(\text{Expr} \times E \times S) \rightarrow (V \times S)$
- Interpreter **implicitly** uses OCaml's runtime stack. The run-time data structure is the call stack containing an activation record for each function invocation - every invocation of interpret is building an activation record on OCaml's runtime - interpreter 2 makes the stack explicit

3.1.1 Code

```

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))

type env = Ast.var -> value

val interpret : Ast.expr * env * store -> (value * store)

val update : env * (var * value) -> env

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)
| 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!âĀĬ")
| 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!âĀĬ)
| Snd e -> (match interpret(e, env, store) with
    | (PAIR (_, v2), store') -> (v2, store')
    | (v, _) -> complain "runtime error. Expecting a pair!âĀĬ)
| 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')
| 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!âĀĬ)
| 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)

```

3.2 Interpreter 2

- Makes the OCaml runtime stack explicit
- Complex values pushed onto stacks
- One stack for values and environments
- One stack for instructions
- Heap used only for references
- Instructions have a tree-like structure

Data Types and Abstract Machine: State is comprised of a heap - global array of values - pair of the form (code, environment-value-stack)

```

type address = int
type value =
  | REF of address
  | INT of int
  | BOOL of bool
  | UNIT
  | PAIR of value * value
  | INL of value
  | INR of value
  | CLOSURE of bool * closure

```

```
and closure = code * env
```

```

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

```

```

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
type state = code * env_value_stack val step : state -> state

let step = function
  (code stack, value/env stack, state) -> (code stack, value/env stack, state)
  | ((PUSH v) :: ds, evs, s) -> (ds, (V v) :: evs, s)
  | (POP :: ds, e :: evs, s) -> (ds, evs, s)
  | (SWAP :: ds, e1 :: e2 :: evs, s) -> (ds, e2 :: e1 :: evs, s)
  | ((BIND x) :: ds, (V v) :: evs, s) -> (ds, EV([(x, v)]) :: evs, s)
  | ((LOOKUP x) :: ds, evs, s) -> (ds, V(search(evs, x)) :: evs, s)
  | ((UNARY op) :: ds, (V v) :: evs, s) -> (ds, V(do_unary(op, v)) :: evs, s)
  | ((OPER op) :: ds, (V v2) :: (V v1) :: evs, s) -> (ds, V(do_oper(op, v1, v2)) ::
    ↪ evs, s)
  | (MK_PAIR :: ds, (V v2) :: (V v1) :: evs, s) -> (ds, V(PAIR(v1, v2)) :: evs, s)
  | (FST :: ds, V(PAIR (v, _)) :: evs, s) -> (ds, (V v) :: evs, s)
  | (SND :: ds, V(PAIR (_, v)) :: evs, s) -> (ds, (V v) :: evs, s)
  | (MK_INL :: ds, (V v) :: evs, s) -> (ds, V(INL v) :: evs, s)
  | (MK_INR :: ds, (V v) :: evs, s) -> (ds, V(INR v) :: evs, s)
  | (CASE (c1, _) :: ds, V(INL v)::evs, s) -> (c1 @ ds, (V v) :: evs, s)
  | (CASE (_, c2) :: ds, V(INR v)::evs, s) -> (c2 @ ds, (V v) :: evs, s)
  | ((TEST(c1, c2)) :: ds, V(BOOL true) :: evs, s) -> (c1 @ ds, evs, s)
  | ((TEST(c1, c2)) :: ds, V(BOOL false) :: evs, s) -> (c2 @ ds, evs, s)
  | (ASSIGN :: ds, (V v) :: (V (REF a)) :: evs, s) -> (ds, V(UNIT) :: evs, assign s a
    ↪ v)
  | (DEREF :: ds, (V (REF a)) :: evs, s) -> (ds, V(deref s a) :: evs, s)
  | (MK_REF :: ds, (V v) :: evs, s) -> let (a, s') = allocate s v in (ds, V(REF a) ::
    ↪ evs, s')
  | ((WHILE(c1, c2)) :: ds, V(BOOL false) :: evs, s) -> (ds, V(UNIT) :: evs, s)
  | ((WHILE(c1, c2)) :: ds, V(BOOL true) :: evs, s) -> (c2 @ [POP] @ c1 @ [WHILE(c1,
    ↪ c2)] @ ds, evs, s)
  | ((MK_CLOSURE c) :: ds, evs, s) -> (ds, V(mk_fun(c, evs_to_env evs)) :: evs, s)
  | (MK_REC(f, c) :: ds, evs, s) -> (ds, V(mk_rec(f, c, evs_to_env evs)) :: evs, s)
  | (APPLY :: ds, V(CLOSURE (c, env)) :: (V v) :: evs, s)
  -> (c @ ds, (V v) :: (EV env) :: evs, s)
  | state -> complain ("step : bad state = " ^ (string_of_interp_state state) ^ "\n")

let rec compile = function
  | Unit -> [PUSH UNIT]
  | Integer n -> [PUSH (INT n)]

```

```

| Boolean b -> [PUSH (BOOL b)]
| Var x -> [LOOKUP x]
| UnaryOp(op, e) -> (compile e) @ [UNARY op]
| Op(e1, op, e2) -> (compile e1) @ (compile e2) @ [OPER op]
| Pair(e1, e2) -> (compile e1) @ (compile e2) @ [MK_PAIR]
| Fst e -> (compile e) @ [FST]
| Snd e -> (compile e) @ [SND]
| Inl e -> (compile e) @ [MK_INL]
| Inr e -> (compile e) @ [MK_INR]
| Case(e, (x1, e1), (x2, e2)) -> (compile e) @ [CASE((BIND x1) :: (compile e1) @
  ↪ leave_scope, (BIND x2) :: (compile e2) @ leave_scope)]
| If(e1, e2, e3) -> (compile e1) @ [TEST(compile e2, compile e3)]
| Seq [] -> []
| Seq [e] -> compile e
| Seq (e :: rest) -> (compile e) @ [POP] @ (compile (Seq rest))
| Ref e -> (compile e) @ [MK_REF]
| Deref e -> (compile e) @ [DEREF]
| While(e1, e2) -> let c1 = compile e1 in c1 @ [WHILE(c1, compile e2)]
| Assign(e1, e2) -> (compile e1) @ (compile e2) @ [ASSIGN]
| App(e1, e2) -> (compile e2) I chose to evaluate arg first
  @ (compile e1) @ [APPLY; SWAP; POP] get rid of env left on stack
| Lambda(x, e) -> [MK_CLOSURE((BIND x) :: (compile e) @ leave_scope)]
| LetFun(f, (x, body), e) -> (MK_CLOSURE((BIND x) :: (compile body) @ leave_scope))
  ↪ :: (BIND f) :: (compile e) @ leave_scope
| LetRecFun(f, (x, body), e) -> (MK_REC(f, (BIND x) :: (compile body) @ leave_scope)
  ↪ ) ::
  (BIND f) :: (compile e) @ leave_scope

```

Driver: idea is that if e passes the front-end and $\text{Interp-0.interpret } e = v$ then driver $(\text{compile } e, []) = v'$ where v' represents v

Evaluating compile e should leave value of e on top of stack
 val compile : expr -> code

```

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

```

3.3 Interpreter 3

- **Flatten code into linear array** Introduce a global array of instructions indexed by a code pointer. At runtime, the code pointer points at the next instruction to be executed:

- LABEL L: associate label L with this location in the code array
- GOTO L: set cp to the code address associated with L

- Add code pointer to machine state
- Adds new instructions: (1) LABEL, (2) GOTO, (3) RETURN
- **Compile away conditionals and while loops**

1. If($e1, e2, e3$)

code for e1
TEST k
code for e2
GOTO m
k: code for e3
m:

2. While(e1, e2)

m: code for e1
TEST k
code for e2
GOTO m
k:

3.3.1 Code

- Code locations are represented as (L, None) - not yet loaded or (L, Some i) - label L has been assigned numeric address i
- Return address is saved in the stack - where i is the location of the closure's code in RETURN

```

let step (cp, evs) = match (get_instruction cp, evs) with
| (PUSH v, evs) -> (cp + 1, (V v) :: evs)
| (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)
| (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)
| (SND, 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 l, evs) -> (cp + 1, evs)
| (HALT, evs) -> (cp, evs)
| (GOTO (_, Some i), evs) -> (i, evs)
| _ -> complain ("step : bad state = " ^ (string_of_state (cp, evs)) ^ "\n")

```

```

let rec comp = function
| Unit -> ([], [PUSH UNIT])
| Integer n -> ([], [PUSH (INT n)])
| Boolean b -> ([], [PUSH (BOOL b)])
| Var x -> ([], [LOOKUP x])
| UnaryOp(op, e) -> let (defs, c) = comp e in (defs, c @ [UNARY op])
| Op(e1, op, e2) -> let (defs1, c1) = comp e1 in let (defs2, c2) = comp e2 in (defs1
  ↪ @ defs2, c1 @ c2 @ [OPER op])
| Pair(e1, e2) -> let (defs1, c1) = comp e1 in let (defs2, c2) = comp e2 in (defs1 @
  ↪ defs2, c1 @ c2 @ [MK_PAIR])
| Fst e -> let (defs, c) = comp e in (defs, c @ [FST])
| Snd e -> let (defs, c) = comp e in (defs, c @ [SND])
| Inl e -> let (defs, c) = comp e in (defs, c @ [MK_INL])
| Inr e -> let (defs, c) = comp e in (defs, c @ [MK_INR])
| Case(e1, (x1, e2), (x2, e3)) ->
  let inr_label = new_label () in
  let after_inr_label = new_label () in
  let (defs1, c1) = comp e1 in
  let (defs2, c2) = comp e2 in
  let (defs3, c3) = comp e3 in (defs1 @ defs2 @ defs3,
    (c1 @ [CASE(inr_label, None)]
      @ ((BIND x1) :: c2 @ [SWAP; POP])
      @ [GOTO (after_inr_label, None); LABEL inr_label]
      @ ((BIND x2) :: c3 @ [SWAP; POP])
      @ [LABEL after_inr_label]))
| If(e1, e2, e3) -> let else_label = new_label () in
  let after_else_label = new_label () in
  let (defs1, c1) = comp e1 in
  let (defs2, c2) = comp e2 in
  let (defs3, c3) = comp e3 in (defs1 @ defs2 @ defs3, (c1
    @ [TEST(else_label, None)]
    @ c2
    @ [GOTO (after_else_label, None); LABEL else_label]
    @ c3
    @ [LABEL after_else_label]))
| Seq [] -> ([], [])
| Seq [e] -> comp e
| Seq (e :: rest) -> let (defs1, c1) = comp e in
  let (defs2, c2) = comp (Seq rest) in (defs1 @ defs2, c1 @ [POP] @ c2)
| Ref e -> let (defs, c) = comp e in (defs, c @ [MK_REF])
| Deref e -> let (defs, c) = comp e in (defs, c @ [DEREF])
| While(e1, e2) -> let test_label = new_label () in
  let end_label = new_label () in
  let (defs1, c1) = comp e1 in
  let (defs2, c2) = comp e2 in (defs1 @ defs2, [LABEL test_label]
    @ c1
    @ [TEST(end_label, None)]
    @ c2
    @ [POP; GOTO (test_label, None); LABEL end_label; PUSH UNIT])
| Assign(e1, e2) -> let (defs1, c1) = comp e1 in
  let (defs2, c2) = comp e2 in (defs1 @ defs2, c1 @ c2 @ [ASSIGN])
| 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
  let result =
    c @ body of program
    [HALT] stop the interpreter
    @ defs in the function definitions
  in result

```

This is a very clean machine, but has a very inefficient treatment of environments - still using OCaml's runtime memory management to manipulate complex values

3.4 Jargon Virtual Machine

Changes:

1. Introduces an **addressable** stack
2. Replaces variable lookup by a location on the stack or heap determined at compile time
3. Adds frame pointers pointing into the stack - so that we can define locations relatively
4. Optimise representations of closures such that they contain only values associated with free variables and pointer to code
5. Restrict values on stack to be simple (int, bool, heap addresses) with complex data being placed onto the heap

3.4.1 Structure

- **Stack Frame**

- Each **frame** has a stack pointer pointing to the top and a frame pointer pointing to the bottom.
 - * Stack elements in virtual machines are generally restricted to be of a fixed size.
 - * Need to shift data from high-level stack to a lower-level stack with fixed size elements
 - * Therefore, puts the data in the heap and place pointers to the heap on the stack
- The frame pointer stays constant in each frame with the stack pointer moving up and down depending on the number of variables in each frame
- As an item calls code at a specific heap address, the top of the previous stack frame contains the pointer to the closure and the argument value with which the new code is called.
- Bottom of the new stack frame contains the saved frame pointer and the return address

- Heap is addressable with a 0-indexed integer byte address
- Separate code array with a code pointer pointing at a specific line of code

3.4.2 Item Code

```

type instruction =
  | PUSH of stack_item
  | LOOKUP of value_path MODIFIED
  | UNARY of Ast.unary_oper MODIFIED
  | OPER of Ast.oper
  | ASSIGN
  | SWAP
  | POP
  | BIND of var not needed
  | FST
  | SND
  | Deref
  | APPLY
  | RETURN

```

```

| MK_PAIR
| MK_INL
| MK_INR
| MK_REF
| MK_CLOSURE of location * int  MODIFIED
| TEST of location
| CASE of location
| GOTO of location
| LABEL of label
| HALT

```

```

type stack_item =
| STACK_INT of int
| STACK_BOOL of bool
| STACK_UNIT
| STACK_HI of heap_index  Heap Index
| STACK_RA of code_index  Return Address
| STACK_FP of stack_index  (saved) Frame Pointer

```

```

type heap_type =
| HT_PAIR
| HT_INL
| HT_INR
| HT_CLOSURE

```

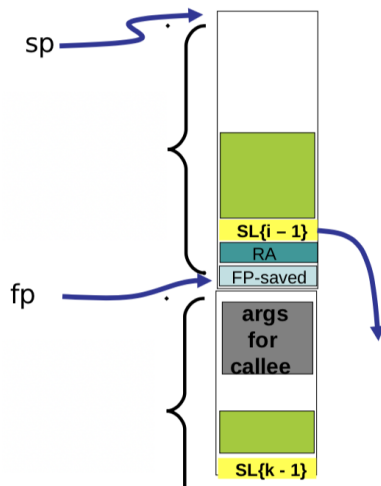
```

type heap_item =
| HEAP_INT of int
| HEAP_BOOL of bool
| HEAP_UNIT
| HEAP_HI of heap_index  Heap Index
| HEAP_CI of code_index  Code pointer for closures
| HEAP_HEADER of int * heap_type  int is number items in heap block

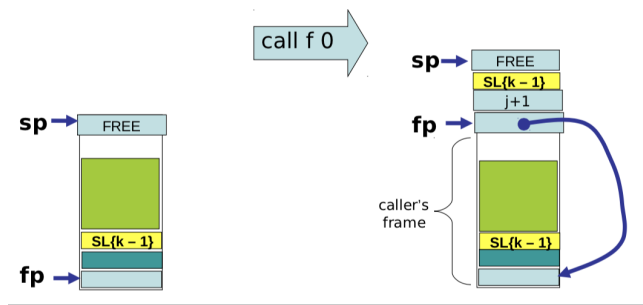
```

3.4.3 Finding variable's value at runtime

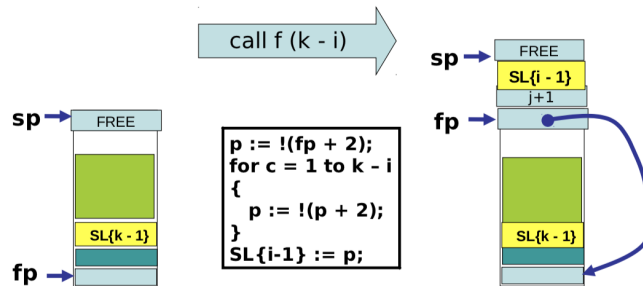
- Every free variable in the body of the closure can be found relative to the frame pointer
- Formal parameter is at stack location fp-2 and other free variables are found at:
 - Follow heap pointer found at fp-1
 - Each free variable can be associated with a fixed offset from this heap address
- **Static Links:**
 - Stack frame for callee defined at nesting depth $i \leq k+1$, with the static link pointing down to the closest frame of definer at nesting depth $i-1$
 - Stack frame for caller defined at nesting depth k used to evaluate code at depth $k+1$



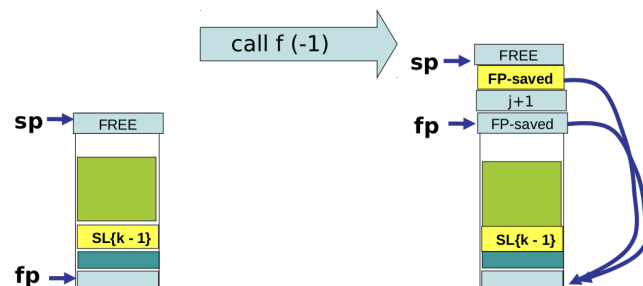
Caller and Callee at same nesting depth k



Caller at depth k and callee at depth $i < k$



Caller at depth k and callee at depth $k+1$



- **Argument values:**

- **Static Distance 0:** pass an integer and the address of the argument is $fp-j$
- **Static Distance d , $0 < d$**

```
p := !(fp + 2);
for c = 1 to d:
```

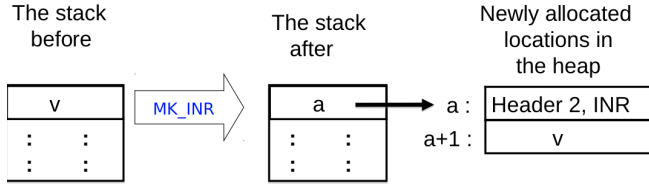
```

    p := !(p+2);
    v := !(p-j);

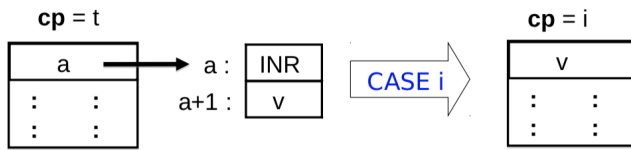
```

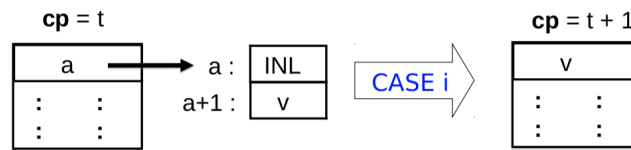
3.4.4 How operations work?

1. MK_INR

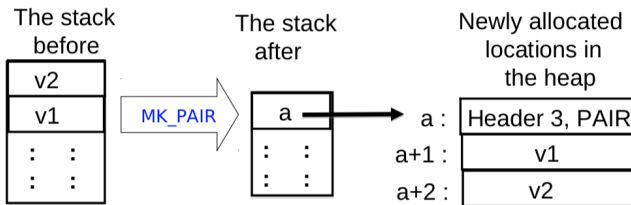


2. CASE

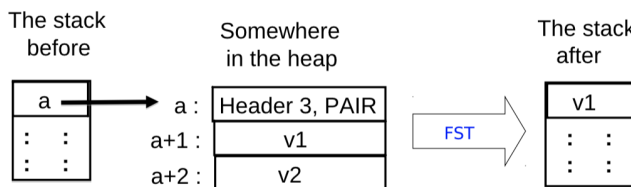
$$(\text{CASE } (_, \text{Some } _), V(\text{INL } v)::\text{evs}) \rightarrow (\text{cp} + 1, (V \ v) :: \text{evs})$$


$$(\text{CASE } (_, \text{Some } i), V(\text{INR } v)::\text{evs}) \rightarrow (i, (V \ v) :: \text{evs})$$


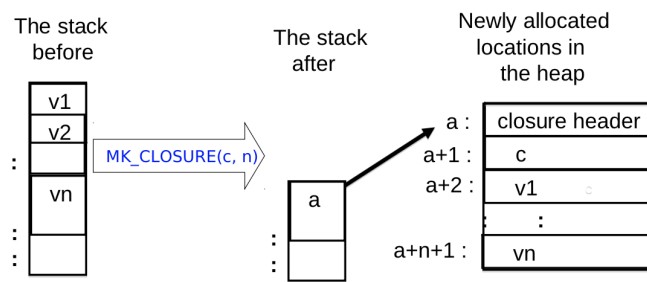
3. MK_PAIR



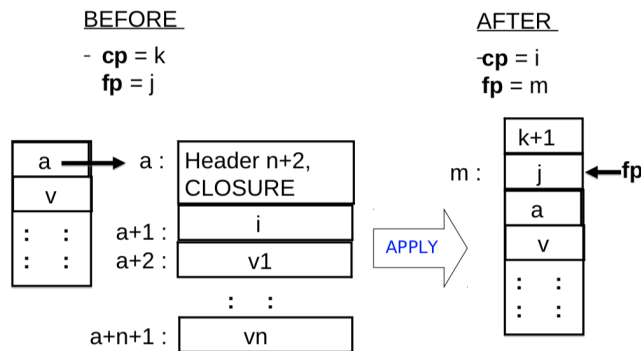
4. FST



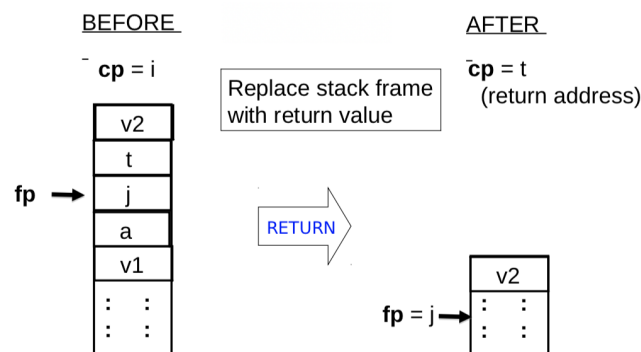
5. **MK_CLOSURE(c, n)**: c is the code location of start of instructions for closure and n is the number of free variables in the body of the closure. Effectively put the values associated with free variables on the stack and then construct the closure on the heap:



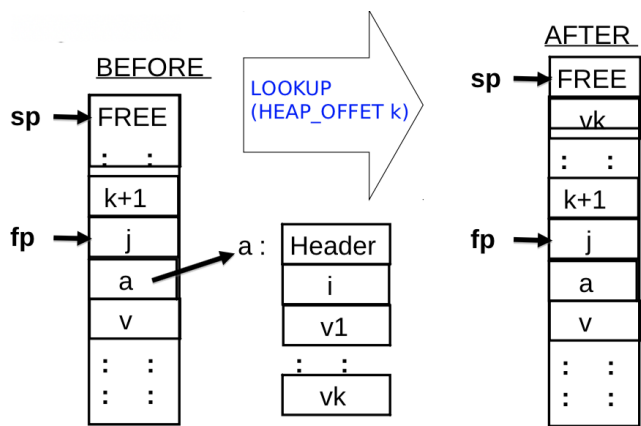
6. APPLY



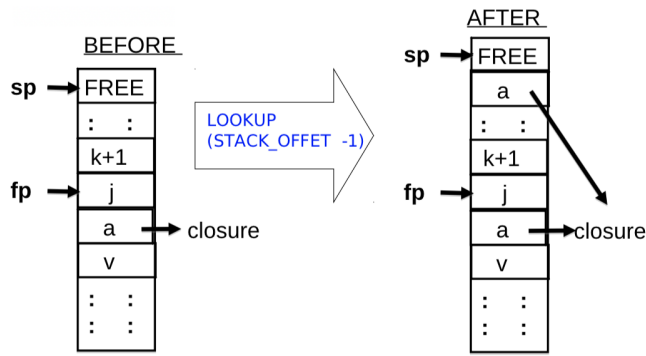
7. RETURN



8. LOOKUP (HEAP_OFFSET k)



9. LOOKUP (STACK_OFFSET - 1) - need to be able to find a closure - has to be a fixed offset from a frame pointer - therefore push the current closure onto the stack



3.4.5 Notes

1. Semantic GAP between L3 and Slang program and a low-level translation has been significantly reduced
2. Implementing the Jargon VM at a lower-level of abstraction looks like a relatively easy problem
3. However, harder to verify compilers and generate efficient code
4. Languages which are higher order - provide more than just first-order functions - are harder. With first-order functions, can avoid allocating environments on the heap since all values associated with free variables will be somewhere on the stack

4 Recursion to Iteration Transforms

4.1 Tail Recursion Elimination

Tail-Recursion: If implemented with a stack, the call stack will simply grow and then shrink - with no ups and downs in between - this code can be replaced by iterative code that does not require a call stack.

This can be done using an OCaml compiler in order to make all tail recursive functions into iterative functions

4.2 Continuation Passing Style (CPS)

This is a general method for transforming any recursive function into a tail recursive function. This works by adding an extra argument (a continuation - represents the rest of the computation). Then, can defunctionalise these continuations and represent them with a stack - therefore have a tail recursive function that carries its own stack as an extra argument.

cnt: function expecting the result of of the main function as its argument

4.2.1 cnt

```
# 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_v2 : (int -> int) * int -> int
let rec fib_cps (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(m - 2, cnt2 a)
      in fib_cps(m - 1, cnt1)
```

Effectively makes the order of the evaluation that is implicit in the original 'fib(m-1) + fib(m-2)' explicit as:

1. compute fib(m-1)
2. compute fib(m-2)

3. add results together
4. return

4.2.2 Identity Continuation

```
let id (x : int) = x
let fib x = fib_cps(x, id)
```

4.3 Defunctionalisation (DFC)

In fib-cps, can replace id, cnt1 and cnt2 with instances of a new data type - therefore remove the need for a functional argument:

```
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)
```

Continuations are just lists, therefore:

```
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, [])
```

4.4 Evaluating Expressions Example

```
type expr =
  | INT of int
  | PLUS of expr * expr
  | SUBT of expr * expr
  | MULT of expr * expr

# 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)

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)

```

Then, defunctionalise that:

```

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)

```

Converting to Stack using Driver and Accumulator

```

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 acc =
  | A_INT of int
  | A_EXP of expr

type cnt_5 = tag list

type state_5 = expr * cnt_5 * acc

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)

```

Actually two independent stacks - one for expressions and one for values: separate these. Can also observe that the evaluator is interleaving the decomposition of the input expression into sub-expressions and the computation of the addition, subtraction and multiplication. It can be refactored into decomposition before computation.

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

```

# low-level instructions
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]

# step_7 : state_7 -> state_7
let step_7 = function
  | (Ipushv::is, vs) -> (is, v::vs)
  | (Iplus :: is, v2::v1::vs) -> (is, (v1 + v2) :: vs)
  | (Isubt :: is, v2::v1::vs) -> (is, (v1 - v2) :: vs)
  | (Imult :: 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, []) 1
```

5 Compilers in OS context

In order to create a Jargon byte code interpreter:

1. Generate a compact byte code for each Jargon instruction
2. Compiler writes the byte codes to a file
3. Implement an interpreter in C or C++ for these byte codes
4. This is much faster at executing than using the ML implementation

5.1 Portability

Can use a Virtual Machine to target multiple platforms - however, may want to compile to an assembler. This lost portability can be regained through the extra effort of implementing code generation for every desired target platform.

5.2 Linker

Assembler: takes symbolic names and addresses to numeric codes and numeric addresses

Object Files: Must contain at least the following (Executable and Linkable Format can be used as a common format for both input and output):

1. Program instructions
2. Symbols being exported
3. Symbols being imported
4. Constants used in the program

1. Takes object files as input - noting all undefined symbols
2. Recursively searches libraries adding ELF files which define symbols until all the names are defined - and throws an error if any symbol is undefined or multiply defined
3. Concatenated all code segments - forming output code segment
4. Concatenates all data segments
5. Creates a single address space by address relocation

Static Linker: Does things at compile time - where all the required input files are included in the executable - this is very assured but may produce very large output files.

Dynamic Linker: Does things at run time - when referring to a shared library, the object files contain stubs, not code and the operating system loads and links the code on demand.

- Executables are smaller
- Library bug fixes do not require re-linking
- Non-compatible changes to library can break previously working programs

5.3 Application Binary Interface

Set of runtime conventions followed by all tools that deal with binary representations of a program, including compilers, assemblers, linkers and language runtime support. Things specified includes:

1. C Calling conventions used for system calls or calls to compiled C code:
 - Register usage and stack frame layout
 - Passing of parameters and how results are returned
 - Caller / callee responsibilities for placement and cleanup
2. Byte-level layout and semantics of object files
 - **Executable and Linkable Format**

Header information; positions and sizes of sections
.text segment (code segment): binary data
.data segment: binary data
.rela.text code segment relocation table: list of (offset,symbol) pairs giving: (i) offset within .text to be relocated; and (ii) by which symbol
.rela.data data segment relocation table: list of (offset,symbol) pairs giving: (i) offset within .data to be relocated; and (ii) by which symbol
...

...
.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

3. Linking, loading and name mangling

5.4 Runtime System

Runtime System is a libraries implementing functionality that is needed to run compiled code on a specific operating system - this is normally tailored to the language being compiled, implementing:

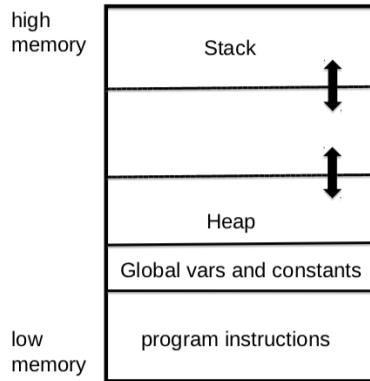
1. Interface between OS and language
2. Memory management
3. Foreign function interface - calling other language code from a piece of code
4. Efficient implementations of primitive operations defined in compiled language
5. May provide runtime type checking, method lookup, security checks, etc

This is used by:

1. **Virtual Machine:** Implementation of the VM will include a runtime system
2. **Linker based system:** Runtime system will feed into the linker alongside the generated code

Implementer of compiler and the runtime system must agree on many low-level details of memory layout and data representation.

5.5 Memory Layout in UNIX



Heap: used for dynamically allocating memory - used for (1) very large objects or (2) for objects returned by functions or procedures and must outlive the associated the activation record. Managed automatically

5.6 x86 Assembler

- CISC architecture with 16, 32 and 64 bit versions
- **32 bit version:**
 - General purpose registers: EAX, EBX, ECX, EDX
 - Special purpose registers: ESI, EDI (often used to pass the first argument), EBP (normally used as the frame pointer), EIP (code pointer), ESP (normally used as the stack pointer)
 - Segment and flag registers
- **64 bit version:**
 - Rename 32-bit registers with R - RAX, ...
 - More general registers: R8, R9, ..., R15
- Two assembler versions - GAS and Intel notation. In GAS, a suffix is used to indicate the width of the arguments: b(yte) - 8 bits, w(ord) - 16 bits, l(ong) - 32 bits, q(uad) - 64 bits

5.6.1 Jargon VM instructions in X86

Jargon VM Instruction	X86 Instruction
GOTO loc	jmp loc
POP	addl \$4, %esp # move stack pointer 1 word
PUSH v	subl \$4, %esp # make room on top of stack movl \$i, (%esp) # where i is integer representing v
FST	movl (%esp), %edx # store "a" into edx movl 4(%edx), %edx # load v1, 4 bytes, 1 word, after header movl %edx, (%esp) # replace "a" with "v1" at top of stack
SND	movl (%esp), %edx # store "a" into edx movl 8(%edx), %edx # vload v2, 8 bytes, 2 words after header movl %edx, (%esp) # replace "a" with "v2" at top of stack

MK_PAIR	<pre> 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 </pre>
---------	---

5.6.2 Arithmetic

Need to be able to distinguish between values and pointers at runtime, in OCaml:

- Store integers as 31 and 63 bits rather than 32 and 64
- Compilers use LSB to distinguish integers (bit=1) from pointers (bit=0)

6 Performance improvements

6.1 Stacks vs Registers

- Stacks based execution is much much slower
- But argument locations are implicit, so the instructions are smaller

But, while registers are fast, there aren't very many, so have to use these registers very effectively:

- Requires careful examination of a program's structure
- **Analysis Phase:** building data structures (directed graphs) that capture the definition
- **Transformation Phase:** using this information to rewrite code, attempting to efficiently utilise registers
- However, the problem is NP-complete

6.1.1 Caller / Callee Conventions

Caller and Callee code can use an overlapping set of registers but an agreement is needed concerning the use of registers.

- Arguments passed in specific registers
- Results returned in specific register
- Must save and restore registers that are used as scratch space
- In general identify subsets of registers as caller saved or callee saved:
 - Caller saved: if caller cares about value in register, save it before making any call
 - Callee saved: caller can be assured that the callee will leave the register intact (if necessary, my saving and restoring it)

6.1.2 Register Spilling

Explains what to do when all the registers are in use, there are a number of options of what to do:

1. Use the stack for scratch space
2. Flush registers
 - (a) Move some register values to the stack
 - (b) Use the registers for computation
 - (c) Restore the registers to their original value

6.2 Inline Expansion

- Avoids building activation records at runtime
- May allow further optimisations
- May lead to code bloat
- Need to be aware that even if inline all occurrences of a function, can't delete definition as it might be required at link time
- Need to be careful about scope of variables

6.3 Constant Propagation - Constant Folding

Propagate constants and evaluate simple expressions at compile-time as far as can be done. These opportunities are often exposed by inline expansion.

6.4 Peephole Optimisation

Sweep a window over the code sequences looking for instances of simple code patterns that can be rewritten to better code - can be employed with constant folding, eg:

1. Eliminate useless combinations (push 0; pop)
2. Introduce machine-specific instructions
3. Improve control flow - rewriting **GOTO L1 ... L1: GOTO L2** to **GOTO L2 ... L1: GOTO L2**

7 OOP object representations

Object is stored as all object data containing an extra argument containing a pointer to the vtable for that class (contains all the methods), where if it is inheriting from something else, the vtable contains information about inherited fields. Therefore, pointer to a child object can be treated as if it is a pointer to a parent object.

vtable

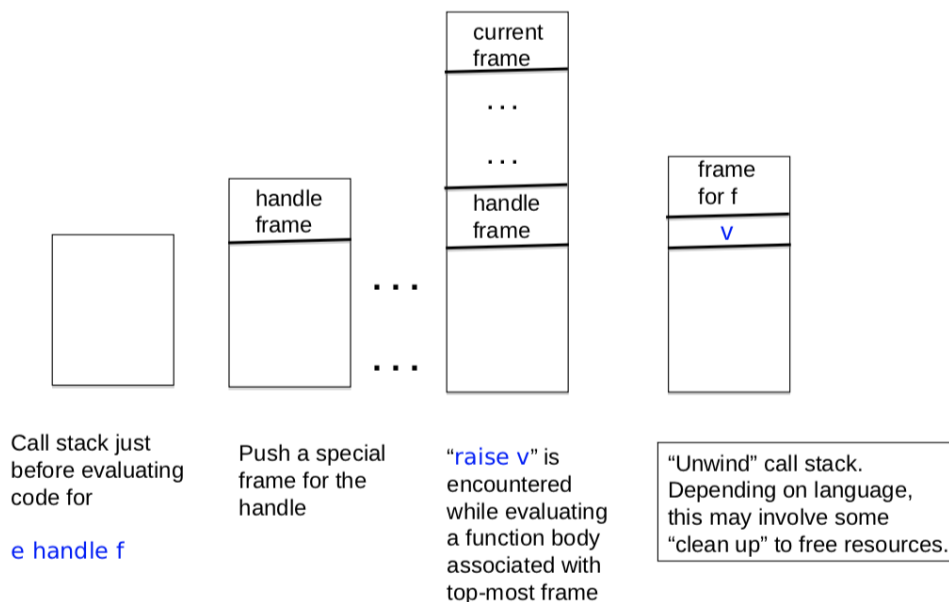
- Method written as methodName_whereDeclared_whereDefined if any methods have been overwritten.
- If nothing has been overwritten, written as methodName_whereDeclaredAndDefined

Need to define whether implementing static or dynamic polymorphism.

8 Exceptions on stack

e handle f: if an expression *e* evaluated 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 the result is (*f v*)

raise e: Evaluate expression to value *v*, then raise *v* as an exceptional value which can only be handled



9 Runtime Memory Management

9.1 Explicit Memory Management

User library manages the memory with the programmer deciding when and where to allocate (`void* malloc(long n)`) and deallocate (`void free(void *adds)`). The library calls the OS for more pages when necessary.

- Gives programmers a lot of control
- Allows us to implement better memory management for higher level language
- Tedious and relies upon humans not making mistakes - not realistic

9.2 Automated Memory Management

Programmers can implicitly allocate new storage dynamically, with no need to worry about reclaiming space no longer used. The memory could be easily exhausted without some method of reclaiming and recycling the storage that will no longer be used.

Deals with: (1) allocation, (2) de-allocation, (3) compaction and (4) memory-related interactions with the OS

Finding what is required: If data is accessible from the root set, then it is not garbage and cannot be removed - couple of methods:

1. **Reference Counting:** Keep a reference count with each object that represents the number of pointers to it - is garbage when the count is 0
 - Very simple and incremental
 - Can't detect cycles
 - Space / time overhead to maintain count
2. **Tracing:** Find all objects reachable from the root set - finding the transitive closure of the pointer graph. Works using copying between two heaps or **mark and sweep**
 - (a) **Mark:** Depth first traversal of object graph from the roots to mark live data
 - (b) **Sweep:** iterate over entire heap, adding the unmarked data back onto the free list

Copying Data: Use two heaps, with one being used by the program and the other being unused until the garbage collection time:

- (a) Start at roots and traverse the reachable data
- (b) Copy reachable data from the active heap to the other heap

- (c) Dead objects are left behind in from space
- (d) Heaps switch roles

- Simple
- Collects cycles
- Automatic compaction eliminates fragmentation
- Twice as much memory used as the program requires
- Long garbage control pauses - does not work for interactive, real time applications
- Long lived objects (which are often larger) have to be copied over and over again

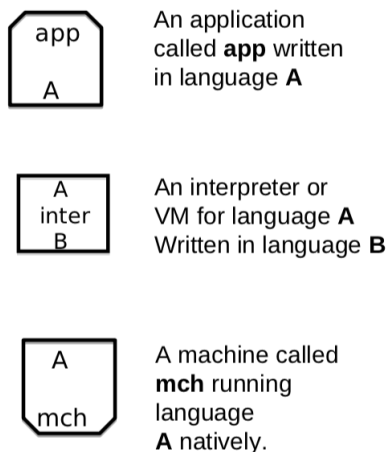
Generational Garbage Collection: Segregate objects into multiple areas by age and do (normal) garbage collection on areas containing older objects less often than the younger ones

1. **Promotion:** promote objects from young generation to old generation when it survives a collection
2. Need to be aware of older objects that are pointing to newer ones
3. Ratio of major and minor collections are often application specific as are the sizes of the generations
4. Often, different GC algorithms used for newer and older generations as they have different characteristics - eg Copying Collection for the new items and mark-sweep for the old items

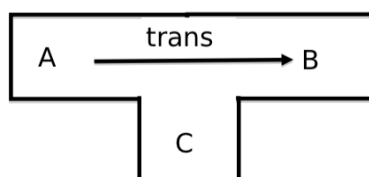
10 Bootstrapping a Compiler

The general idea is to have a compiler compiling itself.

10.1 Notation

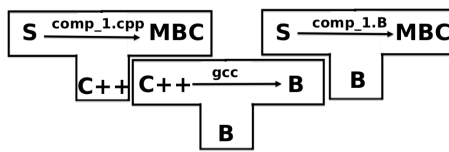


Tombstones: Application a called trans that translates programs in language A into programs in language B and is written in language C

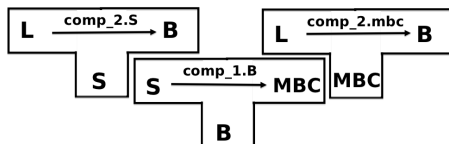


Goal: Created a new language L and have written a compiler in it where the compiler produces code in B (which is a widely used instruction set). Want to compile out compiler so that it can be run on a machine running B.

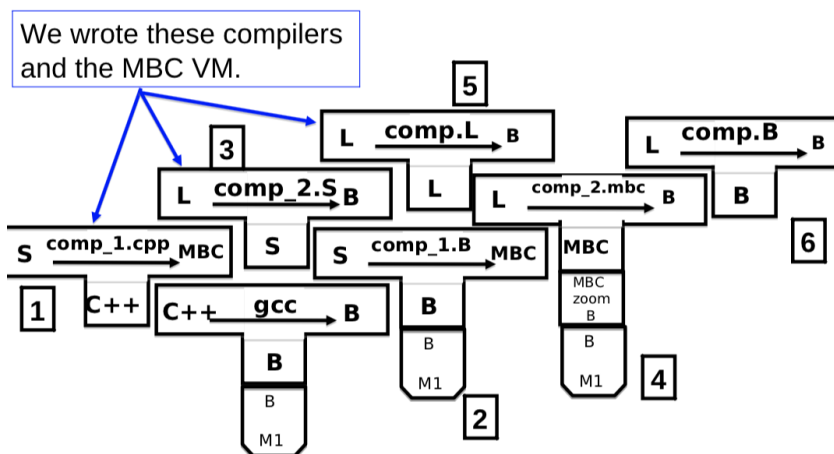
1. Write small interpreter for a small language of byte codes - referred to MBC
2. Pick a small subset S of L and write a translator by hand from S to MBC



3. Write a compiler for L in sub-language S

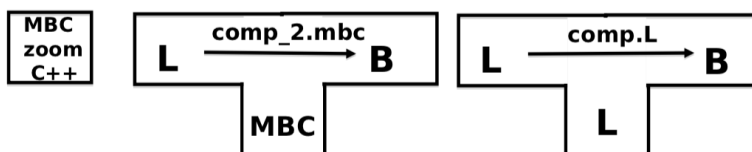


4. Write a compiler for L in L and then compile it: can now use the fully optimised compiler



10.2 Instructions for use

Elements



1. Use gcc to compile the zoom interpreter
2. Use zoom to run comp_2.mbc (can rename it if you want to hide this information) with input comp.L to output the compiler comp.B