Cheatsheet Compilers
from Basil R., Page 1 of 2

## 1 Introduction

### Compiler

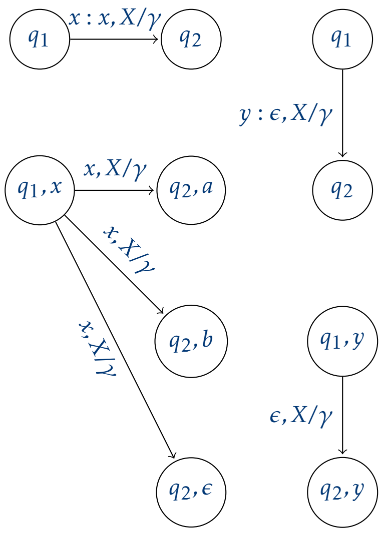
**Front end:** Translates src program into intermediate representation (lexical analysis, scanning; syntactic analysis parsing (type errors reported)). **Backend:** Translates the intermediate representation into the target program (synthesis: translates the abstract representation of the program into target language, target specific optimizations). **Optimizer:** transforms the abstract representation of the code to improve [optional].

### Lexical units, lexemes, tokens

**Lexical unit:** sub-strings and groups created by the scanner (e.g. identifiers, keywords). **Lexeme:** Element of a lexical unit. **Token:** Pair (id=identifier, att=attribute), attribute = extra info.

## 2 Top-down parsing and parser generators

### LPDA




### Prediction

**k-look-ahead PDA**: A k-LPDA is a tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ different: $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \times \Sigma^{\leqslant k} \to 2^{(q \times \Gamma^*)}$. **Configuration change LPDA** The LPDA can move from $(q, auv, X\beta)$ to $(Q', uv, \alpha\beta)$ if there is $(q', \alpha) \in \delta(q, a, X, au)$. $X \in \Gamma$; $a \in \Sigma \cup \{\epsilon\}$; $u \in \Sigma^{\leqslant k-1}$; $v \in \Sigma^*$; and if $|auv| \geqslant k$ then $|au| = k$, else $v = \epsilon$.

### First & Follow

**First:** $First^k(\alpha) = \{w \in T^* : \alpha \Rightarrow^* wx \wedge (|w| = k$ or $|w| < k \wedge x = \epsilon)\}$; **Follow:** $Follow^k(\alpha) = \{w \in T^* | \ \exists \beta, \gamma : S \Rightarrow^* \beta\alpha\gamma \wedge w \in First^k(\gamma)\}$. **First/First:** Intersecting First sets. **First/Follow:** First and Follow sets of variable intersect. **Solutions:** Left factoring First/First; Substitution First/Follow, introduces First/First; Left recursion removal.

### LL(k) grammar

A CFG is LL(K) iff for all pairs of derivations:
$S \Rightarrow *wA\gamma \Rightarrow w\alpha_1\gamma \Rightarrow^* wx_1$
$S \Rightarrow *wA\gamma \Rightarrow w\alpha_2\gamma \Rightarrow^* wx_2$
with $w, x_1, x_2 \in T^*$, $A \in V$, and $\gamma \in (V \cup T)^*$, if $First^k(x_1) = First^k(x_2)$ then $\alpha_1 = \alpha_2$
**Strongly LL(K):** iff for all pairs of rules $A \to \alpha_1$ and $A \to \alpha_2$ with $\alpha_1 \neq \alpha_2$: $First^k(\alpha_1 \cdot Follow^k(A)) First^k(\alpha_2 \cdot Follow^k(A)) = \phi$. *every strong LL(k) grammar is LL(k)*. There is a strict hierarchy.

### Recursive-descent parsers

Instead of pushing sentential forms into a stack, we can make recursive calls to functions handling nonterminals. Consider, e.g., a grammar with rules $S \to Ab$ and $S \to Bc\ldots$

### Attribute grammar

**S-attributed** no inherited attributes. **L-attributed** they allow the attributes to be evaluated in 1 DFS Left-to-right traversal of the derivation tree; $S - att \subseteq L - att$
**L-attribute practice:** An attribute grammar is L-attributed if for all rules $A \to X_1 \ldots X_n$, for all inhertited attributes $\alpha$ of $X_j$ we have that the corresponding semantic rule depends only on: Attributes of $A \to X_1 \ldots X_n$ and inherited attributes of A.

## 3 Scanning with Regular Languages & Parsers with CFG

### Scanner

Splits input in sub-strings and groups in lexical units. Feeds tokenized version of input to parser. **hand-built scanners:** Beyond regular languages; Easier to debug; More efficient.

### Automata

**NFA** $(Q, \Sigma, \delta, q_0, F)$ $Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$ transition func. **DFA** NFA deterministic if $\delta(q, \epsilon) = \phi \ \forall \ q \in Q \wedge |\delta(q, a)| = 1 \ \forall \ (q, a) \in Q \times \Sigma$. For DFAs, $\delta$ form $Q \times \Sigma \to Q$. Can detect word w, in $\mathcal{O}(|w|)$ from regex constructed in P-time.

### Myhill-Nerode theorem

**Equivalence classes:** We say x, y are equivalent with respect to L, written x $\sim_L$ y if xz $\in$ L $\iff$ yz $\forall \ z \in \Sigma^*$ p> **theorem:** Language L is regular iff $_L$ has finite number of equivalence classes. Number of equivalence classes of $_L$ is number of states of smallest DFA.

### recognizing v scanning

**Recognizing:** Given string, simulate DFA and get boolean answer: (not) lexeme recognized. **Scanning:** Given string, scanner returns sequence tokens. Token is longest match (maximal much).

### Longest matches

Find longest prefix of remaining string which is lexical unit. **2 or more matches:** 1. First will be chosen; 2. Disallow, DFAs cant accept same input. Doing: 1. DFAs ordered; 2. Language checking when generating scanner. For all pairs of EREs with $L_1, L_2$, we should check: $L_1 \cap L_2 = \phi$.

### Grammar

G = (V, T, P, S) P: $\alpha \to \beta$ with $\alpha \in (V \cup T)^* V (V \cup T)^*$ and $\beta \in (V \cup T)^*$. **Derivation** Let $\gamma \in (V \cup T)^* V (V \cup T)^*$ and $\delta \in (V \cup T)^*$. We say $\delta$ can be derived from $\gamma$ iff there are $\gamma_1, \gamma_2 \in (V \cup T)*$ and a rule $\alpha \to \beta \in P$ st $\gamma = \gamma_1 \cdot \alpha \cdot \gamma_2$ and $\delta = \gamma_1 \cdot \beta \cdot \gamma_2$.

### Grammar classes

For all rules $\alpha \to \beta$: *class 0*: All grammars no restriction; *class 1*: Context-sensitive grammars: either $\alpha = S$ and $\beta = \epsilon$ or $|\alpha| \leqslant |\beta|$ and S does not appear in $\beta$; *class 2*: Context-free grammars: $\alpha \in V$; *class 3*: $\alpha \in V$ and left-regular $= \beta \in T^* \cup (V \cdot T^*)$. right-regular $= \beta \in T^* \cup (T^* \cdot V)$. **Chomsky's hierarchy:** Languages: $Reg \subset CFL \subset CSL \subset RE$.

### Universality of CNF

**CFG to CNF:** Eliminate: 1. Start symbol from RHS; 2. Rules with nonsolitary terminals. 3. RHS with more than 2 nonterminals. 4. $\epsilon$ productions; 5. Unit rules. **splits:** For grammars in CNF, A can generate w if we can find a rule A -> BC and we can split w in two non-empty words w = u · v st: B -> u and C -> v. It does detect w in polynomial time $\mathcal{O}(|w|^3)$

### Attributes (semantics)

We associate to every nonterminal variable V of the grammar a finite set A(V) of attributes partitioned into synthesized attributes $A_s(V)$ and inherited attributes $A_i(V)$. I Each attribute $\alpha \in A(V)$ has a (potentially infinite) set of possible values. The actual value will be selected based on the appearance of V in the derivation tree. **Attribute mapping functions:** For each rule $\alpha \to \beta_1 \ldots \beta_k$ we have a semantic rule functions $f_{\alpha,j}$ mapping values of certain attributes of $\beta_1 \ldots \beta_k$ to values of $\alpha \in A(\beta_j)$. **welldefinedness:** Semantic rule should not be recursive. They are formulated in a way st all attributes are always defined at all nodes in all possible derivation trees. **synthesized/inherited:** synthesized if its value depends on the vaules of its childen in some derivation tree, if not the inherited.

### (Non)deterministic pushdown automata

A is a tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to 2^{Q \times \Gamma^*}$ transition func. **Deterministic:** 1. $\forall q \in Q$, all $a \in \Sigma \cup \epsilon$, and all $\gamma \in \Gamma$: $\delta(q, a, \gamma)$ has at most 1 element. 2. $\forall q \in Q$, and all $\gamma \in \Gamma$: if $\delta(q, \epsilon, \gamma) \neq \phi$ then $\delta(q, a, \gamma) = \phi$ for all $a \in \Sigma$, but does not accept CFL $L_{pal}$

### Simplifying grammar

**Factoring:** $A \to wB$ and $A \to wC$, split A into $A_1 \to wA_2$ and $A_2 \to B$, $A_2 \to C$. **Indirect left recursion:** $\forall$ pairs of rules $A \to Bw$ and $B \to \beta_1$ we replace the first rule by $A \to \beta_1 w$. Repeated. **Direct left recursion:** If we have rules $(V \to V\alpha_i)_{i \in I}$ and $(V \to \beta_j)_{j \in J}$, we can replace them by $(V' \to \alpha_i)_{i \in I}$, $(V \to \beta_j V')_{j \in J}$, and $V' \to \epsilon$. **unproductive:** A unproductive if there is no word w that can be derived from A. **unreachable:** $X \in T \cup V$ is unreachable if no sentential form contains X.

## 4 MIPS reference

**Arithmetic:** <op> $d, $s, $t: <op> = [add(u)| and| nor| or| sub(u)| xor]; <op> $s, $t, i: <op> = [add((u)i)| andi| ori| xori]; <op> $s, $t: <op> = [div(u)| mult(u)] where result is in hi for div and mod in lo and result for mult in hi and lo;
**Branch:** b[eq|ne] $s, $t, label; b[lez|gtz] $s, label. **Jump:** j[al] label; j[al]r $s
**Load** l[b(u)|h(u)|w] $t, i($s) **datamovement:** m[f|t][hi|lo] $[d|t]

### call codes

**print** 1:int, 2:fl, 3:dbl, 4:str, 11:char; **read** same as before + 1; 10: exit

### stackframe

**building** sw $fp, 0($sp); move $fp, $sp; subu $sp, $sp, <size> + 4; sw $ra, -4($fp); sw <reg_to_store>, -offset($fp)
**destructing** lw <reg_to_load>, -<size> + 4($fp); lw $ra, -4($fp); move $sp, $fp; lw $fp, 0($sp); j $ra

## 5 LLVM Reference

### Type

i{1-32}: int; float, double, fp128: float; <type>*: pointers

### Function

define <type> @<f_name>(<type> %<name>, ...) {
...instructions...
}
call <type> @<f_name>(<type0> <arg1>, ...)

### Labels

<label>:
...instructions...
br label %<label>
br i1 <cond>, label <true>, label <false>

### Comparisons

{i,f}cmp <policy> <type> <arg1>, <arg2>; i-policies: eq, neq, (u,s)gt, (u,s)lt, (u,s)le, (u,s)ge [u=unsigned, s=signed]; f-policies: oeq,ogt,oge,olt,ole,one,ord,ueq,ugt etc[o=ordered,u=unordered]

### Phi

phi <type> [<val0>, <label0>], ...: based on predecessor block

### Arithmetic

<op> <type> <arg1>, <arg2>: {f}add, {f}sub, {f}mul, {u,s,f}div, and, or, xor;{f,s,u}rem, ashl, lshr, ashr

### Load/Store

alloca <type>: allocate space on the stack, ret=ptr; store <val_type> <val>, <ptr_type> <ptr>, ret=void; load <val_type>,<ptr_type> <ptr>, ret=val

### Conversion/Cast

{trunc, zext, sext, fptrunc, fpext} <in_type> <in> to <out_type>: ret=out

### Arrays

type = [<size> × <type>]; alloca; getelementptr <arr_elem_type>, <arr_type_ptr> <ptr_to_array>, <idx_type> 0, <idx_type> <idx> (arrptr[0][idx] == arr[idx])

### Strictly dominates & dominance frontier

## 6 Good code generation

### Usage and liveness

**Usage:** Let i < j. The value of x computed at i is used at j. **Liveness:** Let $i < k$. If value of x computed at i is used at j then x is live at all $i \leqslant j < k$.
**Algorithm:** Instructions block from last to first. $\forall$ instructions $i : x = yopz$:
1. Attach to instruction i the usage and

liveness information of x, y, z. 2. Set x to not live and no next use. 3. Set y, z to live and the next uses of y, z to i. **Datastructure:** Contains additional information in the symbol table, for the algorithm. Or table per basic block. Usage and Liveness per instruction.

### Minimizing load and store

**Register descriptors:** Variable names whose current value is in a register (per register). **Address descriptors:** All locations where the current value of a variable is stored. **Algorithm:** $\forall$ instruction $i : x = yopz$ we do the following. 1. Select registers $R_x, R_y, R_z$ for the var using $getReg(x = yopz)$. 2. If y is not loaded in $R_y$ then issue instruction. 3. If z is not loaded in $R_z$ then issue instruction. 3. Issue instruction $R_x = R_y opR_z$ at end of block all marked live -> store instruction. Copy no machine-instruction output if $R_x == R_y$

**Algorithm reg for operands:** $R_y$ can be chosen: 1. If y in reg r then $R_y = r$ 2. If y is not in reg but r is empty then $R_y = r$. 3. r is candidate: If all var descriptor says their value is in r also have another location then return r; If only var whose descriptor says value is in r is x and $x \notin \{y, z\}$ then return r; If all var not used ofeter instruction return r; Spill the vars into memory. **result reg:** reg holding r or those of x,y if not live hereafter.

### Peephole

Sliding window, replaces instructions. Optimizes: redundant-instructions; Flow-of-control; Algebraic; Machine idioms (auto-increments for addresses).

### Systematic spilling

Infinite registers generate its code; Construct a register-interference graph (=k-colouring of graph). Where nodes are registers, vertices are nodes alive together.

### Sethi-Ullmann

**equal-children:** recursively gen code right child base=b+1, result in $R_{b+k-1}$; do the same for left child with base b result in register one lower; result in $R_{b+k-1}$. **different-children (m<k):** big child same; result small child in $R_{b+m-1}$; Answer in big child register. **With spilling:** Big child chosen; Recursive, use b=1, result in $R_r$ then in memory; generating for small child; If label >r use base 1 else b=r-j. Recursive, result in $R_r$; Big child mem in $R_{r-1}$; compute in $R_r$

### Pipelining

1. Instruction fetch 2. Instruction decode and register fetch 3. Execute 4. Memory access 5. Register write back

## 7 Bottom-up parsers

### Shift & Reduce

It builds a right-most derivation in reverse order. **Reduce/Reduce:** Top corresponds to handle for 2 different rules. **Shift/Reduce:** Top corresponds to handle, but parser can shift too.

### PDA bottom-up

Shift: For all terminals a, the PDA has a transition that reads a and pushes a into the stack. Reduce: For every rule $A \to \alpha_1 ... \alpha_n$ the PDA has: states $A, \alpha_1, ..., \alpha_i$ $\forall 1 \leqslant i < n$ and $(A, \epsilon)$; transition pops letters from $alpha^R$ from the stack without reading from the input. A transition from $(A, \epsilon)$ to push A without reading from input.

### Sentential-form prefixes on the stack

$(q, w, \gamma Z_0)$ with q not of form $(A, ...)$ then: $S \Rightarrow^* \gamma^R \cdot w . \gamma^R$ happens to be a **viable prefix**. All the viable prefixes are recognized by the CFSM.

### Generalized algorithm action table:

Same CFSM for LR(0), SLR(K), LALR(k) **LR(0)** 1. Push the initial state 0 of the CFSM into the stack S. 2. As long as we can't accept or get an error: 2.1. If T (top(S)) is Shift, then put the next symbol in variable c. 2.2. Otherwise, if T(top(S)) is {j}. 2.2.1. pop $|\alpha|$ times from S, where $A \to \alpha$ is the j-th rule. 2.2.2. and put A in c. 2.3. Let q' be the next state $\delta(top(S), x)$ of the CFSM after reading c. 2.4. Push q' into S. **SLR(k)** 2.1. append "and the look-ahead starts with a, c=a". 2.2. Append "$A \to$ is the j-th rule, and the look=ahead l is in $Follow^k(A)$", lookahead combos from those sets become columns. **LALR(k)** same as SLR(k). But uses ItemFollow. **LALR(1) propagation graph** $(V, E, l)$ V contains all state-item-pairs (s,u) st $u \in s$; E contains $(\langle s,u\rangle, \langle s,u\rangle)$ iff 1. t is the closure of the a-successor of s, $u = A \to \alpha_1 \cdot a\alpha_2$, and v = $u = A \to \alpha_1 a \cdot \alpha_2$. 2. $s = t$ and $u = A \to \alpha_1 \cdot B\alpha_2$, $v = B \to \beta$, $\alpha_2$ can produce $\epsilon$; The labelling function $l : V \to 2^{\Sigma^{\leqslant 1}}$ is st: $l(\langle (s, B \to \cdot \beta)\rangle) = \cup\{First^1(\alpha_2) | \langle s, A \to \alpha_1 \cdot B\alpha_2\rangle \in V\}$ and all other vertices are mapped to the empty set.

**ItemFollow:** For a vertex v, $ItemFollow^1(v)$ is the set $\cup\{l(u) | u\ has\ a\ path\ to\ v\}$ so this tells us whther we caqn apply reduction rul $A \to \beta$ from a CFSM state s based on the $ItemFollow^1(v)$ of $v = (s, A \to \beta \cdot)$

### LR(k)-CFSM

**LR(k)-items:** $(A \to \alpha_1 \cdot \alpha_2, w)$ where $w \in \Sigma^{\leqslant k}$ **LR(k)-closure:** The closure of $(A \to \alpha_1 \cdot \alpha_2, w)$ is the set of all LR(k)-items $(B \to \cdot \beta, y)$ where: 1. $B \to \beta$ grammar rule 2. $y \in First^k(\alpha_2 w)$; It includes I and is minimal set. **CFSM states:** states of the CFSM are now subsets of LR(k)-items. The a-succeswsor of a state I, for $a \in T \cup V$, is the closure of the set $\{(A \to \alpha_1 a \cdot \alpha_2, w) \mid (A \to \alpha_1 \cdot a\alpha_2, w) \in I\}$ **LR(k)-CFSM:** $q_0$ is the closure of $\{(A \to \alpha_1 a \cdot \alpha_2, w) \mid (A \to \alpha_1 \cdot a\alpha_2, w) \in I\}$

### Action tables

**LR(0)** Index grammar rules $1 \leqslant j \leqslant n$ and states from the CFSM $0 \leqslant i \leqslant k$ The table T maps each i to a set of actions: 1. $T(i)$ contains (a Reduce) j if state i has the item $A \to \alpha \cdot$ with $A \to \alpha$ the j-th rule. 2. T(i) contains Shift if state i has an item $A \to \alpha_1 \cdot \alpha_2$. 3. T(i) contains Accept if state i has an item $S \to \alpha \cdot$. 4. $T(\phi)$ contains an error action only. **SLR** replace (a Reduce) by *nothing*, replace *Shift* by *(a Shift) a* and prepend an a to $\alpha_2$ **LR(k):** add and look-ahead l to a set of actions to the 1st line; change to on 1 to $A \to \alpha \cdot, l)$; change to on line 2 $(A \to \alpha_1 \cdot \alpha_2, y)$ and $l \in First^k(\alpha_2 y)$.

### Conclusions

$\forall k \geqslant 1$, the LR(k) languages, LR(1) languages, and languages recognizable by DPDA are all the same. (**Most** languages use LALR(1) grammar)

## 8 Good code generation

### Usage and liveness

**Usage:** Let i < j. The value of x computed at i is used at j. **Liveness:** Let $i < k$. If value of x computed at i is used at j then x is live at all $i \leqslant j < k$. **Algorithm:** Instructions block from last to first. $\forall$ instructions $i : x = yopz$: 1. Attach to instruction i the usage and liveness information of x, y, z. 2. Set x to not live and no next use. 3. Set y, z to live and the next uses of y, z to i. **Datastructure:** Contains additional information in the symbol table, for the algorithm. Or table per basic block. Usage and Liveness per instruction.

### Minimizing load and store

**Register descriptors:** Variable names whose current value is in a register (per register). **Address descriptors:** All locations where the current value of a variable is stored. **Algorithm:** $\forall$ instruction $i : x = yopz$ we do the following. 1. Select registers $R_x, R_y, R_z$ for the var using $getReg(x = yopz)$. 2. If y is not loaded in $R_y$ then issue instruction. 3. If z is not loaded in $R_z$ then issue instruction. 3. Issue instruction $R_x = R_y opR_z$ at end of block all marked live -> store instruction. Copy no machine-instruction output if $R_x == R_y$

**Algorithm reg for operands:** $R_y$ can be chosen: 1. If y in reg r then $R_y = r$ 2. If y is not in reg but r is empty then $R_y = r$. 3. r is candidate: If all var descriptor says their value is in r also have another location then return r; If only var whose descriptor says value is in r is x and $x \notin \{y, z\}$ then return r; If all var not used ofeter instruction return r; Spill the vars into memory. **result reg:** reg holding r or those of x,y if not live hereafter.

### Peephole

Sliding window, replaces instructions. Optimizes: redundant-instructions; Flow-of-control; Algebraic; Machine idioms (auto-increments for addresses).

### Systematic spilling

Infinite registers generate its code; Construct a register-interference graph (=k-colouring of graph). Where nodes are registers, vertices are nodes alive together.

### Sethi-Ullmann

**equal-children:** recursively gen code right child base=b+1, result in $R_{b+k-1}$; do the same for left child with base b result in register one lower; result in $R_{b+k-1}$. **different-children (m<k):** big child same; result small child in $R_{b+m-1}$; Answer in big child register. **With spilling:** Big child chosen; Recursive, use b=1, result in $R_r$ then in memory; generating for small child; If label >r use base 1 else b=r-j. Recursive, result in $R_r$; Big child mem in $R_{r-1}$; compute in $R_r$

### Pipelining

1. Instruction fetch 2. Instruction decode and register fetch 3. Execute 4. Memory access 5. Register write back

## 9 Type checking and other semantic analyses

### Type checking

Assigning a type t to a variable x is, in essence, an invariant. Because languages are turing complete undecidable. Conservative if static, applies inference rules to deduce types of expression (without caring about reachability). We can use implicit casting if we have expressions of the type 4 + 1.0. The hierarchy is saved in memory. **Typechecker:** Verify the validity of the expression given the types of the descendants. Derive the type of parent expressions (propagation). Via bottom-up traversal of the AST using the symbol table. Statements have no type, so they start the recursive traversal of their

### General reachability

Undecidable, halting problem. Conservative approach. Focus list of statements, reachable and terminate normally, ie control flow "falls through". **Algorithm:** Consider a statement list S = s 1 , . . . , s n ; 1. If S is reachable then s 1 is reachable. 2. If s n terminates abnormally then so does S. 3. If S is the body of a function or procedure then S is reachable. 4. Local variable declarations, assignments, function calls, memory allocation, increment, decrement instructions all terminate normally. 5. A null statement or statement list does not generate error messages if it is not reachable. However, they propagate their non-reachability status to their successors. 6. If a statement has a predecessor then it is reachable if its predecessor terminates normally. For reachability analysis we assume the conditioncan be both true and false. *while loop* If it evaluates constant true -> the while-statement is marked as abnormally terminating — since it is an infinite loop. false -> unreachable; If body contains break then normal terminating (when was abnormal). If body terminates normally, then the while statement terminates normally (unless infite with no break).