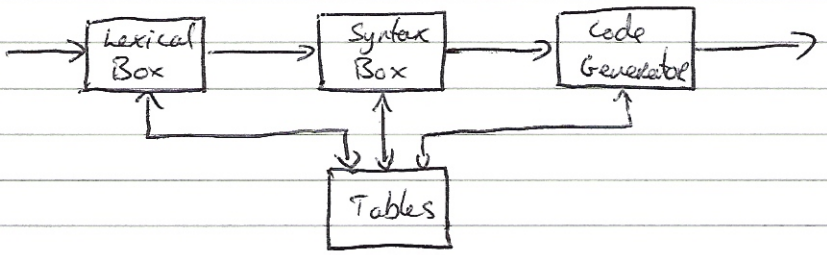
CS3071: Compiler Design Chapter 1

- **Language Processors**: the computer programs that enable the computer to "understand" the commands and inputs of the human, there are 2 types:

  - **Interpreter**: program that accepts a program written in its "source language" and performs the computations implied

  - **Translater**: program that accepts a program written in its "source language" and outputs another version of this program in its "output language", there are 2 types
    (usually the machine language of some computer)
    - **Assemblers**: Translate low-level languages
    - **Compilers**: Translate high-level languages

- **Simple Compiler Model**



- **Tables**: store long-term / global info
  e.g. **Symbol Table**: stores info about each variable or identifier

- **Lexical Box**: breaks up the string of characters input into "tokens" each with a "class" (e.g Constant) and a "value" (e.g. 13).
  * some tokens may not require a value

- **Syntax Box**: translates sequence of tokens from lexical box into another sequence that more closely represents the order of operations the programmer intended. Takes "tokens" and creates "atoms", order of atoms is order things to be carried out.
  e.g.
  A + B * C ⇒ MULT(B, C, R1) ADD(A, R1, R2)
  (A, +, B, *, C: tokens; MULT(B, C, R1), ADD(A, R1, R2): atoms)

- **Code Generator**: expands atoms from sytax box into sequence of computer instructions with some intent
  - Often performs Sophisticated analysys of runtime contents etc. to pick best regs
  - Semantic Processing: decisions on 'meaning' of symbols sometimes done by Semantic box
  - Optimisation = compiler subactivities which are not strictly speaking necessary

- Passes and Boxes:
  - One-pass Compiler: control bounced between boxes as atoms and tokens requested and produced. Reads source code in "one-pass"
  - Three-pass Compiler: lexical box prepares complete file of tokens and so on.
- Run-time Implementation: of a language is the set of data structures and control mechanisms that need to exist at run-time to implement the features of the language

CS3071: Compiler Design Chapter 5: Pushdown Machines

- **Pushdown Machine**: essentially FSM with stack available to it (aka the "Control")
  - Each step determined by set of rules dependent on the state, the top stack symbol, the current input symbol
  - **Control**: selects either an exit OR a transition
    - **Stack Operations**: - push a symbol to the stack
      - pop top stack symbol
      - leave unchanged
    - **State Operations**: - Change to a specified state
    - **Input Operations**: - Get next input symbol and make it the current input symbol
      - Retain present input symbol as current

  - **Pushdown Machine Definition:**

(Primitive) **Strings**:
- finite set **input symbols** incl. "$\dashv$"
- " " **stack symbols** incl. "∇"
- " " **states** incl. starting state
- **Control**: assigns exit / transition to each combination of input symbol, stack symbol, state. Non-exit transitions are a **stack op**, a **state op** and an **input op**
- **Starting Stack**: "∇" followed by a sequence of stack symbols

  - **Pushdown Recogniser**: Pushdown Machine with two exits accept and reject.

  - **Transitions**:
    - POP: Pop top stack symbol
    - PUSH(A): push symbol A to top of stack
    - STATE(s): state s is the next state
    - ADVANCE: take next input to be used as current input
    - RETAIN: Retain current input.

NB: Retain stack is absence of POP/PUSH(A)
STATE(s) omitted if exactly 1 state.

*If multiple states draw a control table for each state.

- **Sets of Sequences Notation:**
  - **Union:** $P \cup Q$ / $P + Q$ (on sets of sequences) e.g. $\{1,2\} + \{3,2\} = \{1,2,3\}$
  - **Concatenation:** $P \cdot Q$ (of sequences) e.g. $\{10\}\{1,00\} = \{101,1000\}$ *(can usually leave out dot)
    - Concatenation of a set with itself:
      - $R \cdot R$ or RR or R^2
      - $R^0 = \{\varepsilon\}$
  - **Kleene Star:** A^* e.g. $\{1,0\}^* = \{\varepsilon, 1, 0, 10, 01, 11, 100, \ldots\}$
    - $R^* = R^0 + R^1 + R^2 + \ldots$
    - **Klean Plus:** $A^+ = AA^* = A^*$ excluding ε
  - **Extensions:** • $\{1^n, 0^n\}$ $n > 0$ $= \{10, 1100, \ldots\}$
    • $(abc)^r = cba$
- **Extended Stack Operations:**
  - **REPLACE (A B C):** Pops top of stack and pushes A, B, C in turn
    e.g. $\nabla XYZ \Rightarrow \nabla XYABC$
    * can use this to avoid extra states by maintaining stack symbol at top of stack
- **Pushdown Translator:** Pushdown machine that produces an output sequence.
  - **Output Operation:** OUT (AB) outputs the sequence AB
- **Cycling:** Should be avoided if procedures followed correctly

CS3071: Compiler Design: Chapter 7: Syntax-Directed Processing

- **Translation Grammars**: A context free grammar where set of terminals symbols is partitioned into a set of *input symbols* and a set of *action symbols*
  - **Activity Sequence** = string in the language specified by a translation grammar
  - **Translation**: set of pairs of "input parts" and "action parts" of the activity sequences of a translation grammar.
  - **Input Grammar**: Translation Grammar mines action symbols
    - **Input Language**: Language specified by this grammar
  - **Action Grammar**: Translation Grammar mines input symbols
  - **Synthesised Attributes**: passed up the derivation tree.
    - i.e. synthesised from non-terminals and terminals below
    - & have rules assigning LH symbols
  - **Inherited Attributes**: passed down the derivation tree
    - i.e attributes inherited from symbols above them (as different productions are called)
    - & have rules assigning RH Symbols

- **Attributed Translation Grammars**: Translation Grammar for which:
  1. Each input, non-terminal and action symbol has a finite set of attributes, each with a possibly infinite set of values permissable
  2. Each non-terminal and action symbol, attr, is either inherited or synthesised.
  3. **Rules for inherited attrs**:
     a. for each inherited attr on the RHS of a production there is a rule saying how to compute it as a function of other attrs in the production
     b. each inherited attr of the start symbol has initial value specified
  4. **Rules for synthesised attrs**:
     a. for each synth attr. on the LHS of a production there is a rules stating how to compute it as a function of other attrs in the production
     b. for each synth attr of an action symbol there is a rule stating how to compute it as a function of other attrs of the action symbol.

- Attributed Derivation Trees: Construction Rules
  1. Use underlying unattributed translation grammar to construct a derivation tree for an activity sequence of unattributed input and action symbols
  2. Assign attr values to input symbols in derivation tree
  3. Initialise start symbol in derivation tree with initial inherited attrs
  4. Compute values of attrs of symbols in tree by repeatedly: taking an attr not yet on tree with arguments to rule assigning it already on the tree and compute and add its value to the tree

  * Tree is "complete" if step 4 assigns a value to every attr of every symbol in tree.

  * Grammar "well defined" if above steps always lead to a complete tree.

- Attributed Activity Sequence: activity sequence whose input and action symbols have attrs

- Ambiguous Grammar: has more than one derivation tree for at least one string in its language (avoid)

CS3071: Compiler Design: Top-Down Processing: Chapter 8

- **Parser**: Processor whose operation involves recognition of productions
- **Intermediate String in Leftmost Derivation**: post input sequence concatenated with the stack.
  * Leftmost non-terminal is always parsed first
- **REPLACE (<A><B><C>)**: uses top symbol of Right convention
- **S-Grammars**: One class of grammars for which top-down pushdown recognisers can always be found
  - **Context-free Grammar is an S-grammar if**:
    1. RHS of each production begins with a terminal symbol
    2. If 2 productions have same LHS they begin with different terminal symbols on the RHS
- **One-state pushdown Recognizer for S-Grammar**:
  1. **Input set**: is terminals of Grammar and ⊣
  2. **Stack Symbols**: ▽, non-terminals of grammar, any terminals appearing in RHS of a production and not on extreme left
  3. **Starting Stack**: ▽ <S>
  4. **Control**: one-state control table with stack symbols labelling the rows and input symbols labelling columns
  5. **Table Entries**: If for each production, <A> → b α corresponds to REPLACE (α^R), Advance and <A> → b corresponds to, POP, Advance
  6. **Table Entries for terminal (b)** in row b column b are POP, ADVANCE
  7. ▽, ⊣ = ACCEPT
  8. All others = Reject

  * If a language has an S-Grammar it can be recognized by a one-state pushdown machine using REPLACE
- **Top-down Processing of Translation Grammars**:
  - When an action symbol(s) symbol at top of stack OUT(x), POP, RETAIN

Top-Down Processing of Translation Grammars:

- General Rules for Top-Down Pushdown Translator for translation grammar whose input was an S-Grammar:
  1. Input Set: Terminals of grammar and ⊣
  2. Stack Symbols: Non-terms, inputs on RHS of production but not beginning it, ▽
  3. Starting Stack: ▽
  4. Control: One-state control table: stack → rows, Input → columns
  5. Table Entries: $\langle A \rangle \to \xi\, b\, \psi\, \alpha \Rightarrow OUT(\xi\psi), REPLACE(\alpha^R), ADV$
     (ξ: possibly null action symbol sequence; b: term; $\xi\psi$: string corresp to $\xi\psi$ action symbols; α: sequence of action, terms and non terms not beg in with action)
  6. b is terminal: then for $(b, b) \Rightarrow$ POP, ADVANCE
  7. $(\triangledown, \dashv) \Rightarrow$ ACCEPT
  8. If $\{x\}$ stack symbol: All entries are OUT(x), POP, RETAIN
  9. All other entries REJECT

* Given string translation grammar whose input is an S-grammar, it can be performed by one-state pushdown machine using REPLACE

- Q-Grammar: More general class of grammar that can be recognized by top-down pushdown machines
  - Follow Set: FOLLOW($\langle x \rangle$), Given context free grammar with start symbol $\langle S \rangle$ and nonterm $\langle x \rangle$ is:
    The set of input symbols that can immediately follow on $\langle x \rangle$ in any intermediate string derived from $\langle S \rangle \dashv$

*∴ Useless to apply $\langle x \rangle \to \epsilon$ for inputs not in the follow set of $\langle x \rangle$

- Selection Set: inputs for which a pushdown control must apply a production:
  - SELECT($\langle A \rangle \to b\, \alpha$) = {b}
  - SELECT($\langle A \rangle \to \epsilon$) = FOLLOW($\langle A \rangle$)

- Context-free Grammar is Q-Grammar iff:
  1. RHS of each production either ϵ or begins with terminal
  2. Productions with same LHS have disjoint selection sets

- Rules for push down control same as s-Grammar but I added:
  - $\langle A \rangle \to \epsilon \Rightarrow$ POP, RETAIN

*If a language has a Q-grammar can be recognized by top-down push down

CS3071: Compiler Design: Chapter 8: Top-Down Processing

- **LL(1) Grammars:**
  - **First Set:** given context free grammar and intermediate string α FIRST(α) is: the set of terminals that can occur at the beginning of intermediate strings derived from α
  - **Nullable:** · String α nullable iff: $\alpha \stackrel{*}{\Rightarrow} \epsilon$
    · Production nullable iff: RHS is nullable
  - **More General Selection Set:**
    · **For Not Nullable α:** SELECT($\langle A \rangle \rightarrow \alpha$) = FIRST($\alpha$)
    · **For Nullable α:** SELECT($\langle A \rangle \rightarrow \alpha$) = FIRST($\alpha$) ∪ FOLLOW($\langle A \rangle$)
  - **Context-free Grammar is LL(1) iff:** Productions with same LHS have disjoint selection sets
  - **Rules the same as S-Grammars except:**
    · $\langle A \rangle \rightarrow \alpha$ ⇒ REPLACE(α^R), RETAIN, where α does not begin with a non-terminal
    · If Nullable production for $\langle A \rangle$ and no entry for stack $\langle A \rangle$ and input is implied by above Rule: then either nullable production applied or Reject.

- **Error Processing in Top-Down Parsing:**
  - **Non sentences:** Input sequences not in the language
  - **Error Recovery:** when compiler modifies configuration of pushdown machine, or hitting bottom, so it can continue processing input and produce more error messages if necessary.
  - **Offending Symbol:** current input when Recogniser Rejects
  - **Prefix Property:** (if recognizer has this it detects an error at the earliest possible input) had by pushdown recognizer if: for all nonsentences the sequence to the left of the "Offending Symbol" is the prefix of some acceptable input string.
  - **Begin Error Processing by Choosing Error Messages for each of the reject table entries**
  - **Error Recovery Routines:** adjust input and stack (outputing error messages) and continue as if no error occurred

* Can make incorrect assumptions

- Error Processing in Top-Down Parsing
  - "Local" Approach: fill each reject entry with ordinary stack and input operations. If particular error detected, error message produced, stack and input ops are performed, then processing resumes
    - If single input symbol produces several messages, only first printed
  - "Global" Approach: scan ahead until "trustworthy" symbol found, then recovery depends on that symbol and top of stack
    (used when insufficient confidence in recovery routines or mem requirements don't allow recov routines for each entry)
    - Trustworthy Symbols:
      - Synchronising Symbols: symbol for which designer has reasonable confidence they know how to restart the processing e.g. (END, THEN)
        When scanning routine finds one, stack popped until Synch Symbol and top stack symbol can correctly be applied
      - Beginning Symbols: symbol whose occurrence enables designer to predict several of next nonterminals / terminals but may not be able to restart processor
        Push {ERROR} onto stacks followed by predicted symbols. If {ERROR} becomes top of stack prediction correct, {ERROR} popped and continues scanning for Synch/Begin symbols
- Recursive Descent:
  - Basic Idea: Procedure for each non-term of the grammar that recognizes that non-term that call eachother where appropriate.

* Work out select sets, verify grammar LL(1), write recursive descent for each non-terminal.

CS3071: Compiler Design: Chapter 9: Top-Down Processing of Attributed Grammars

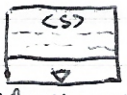
- **L-Attributed Grammars:**
  - Attributed translation grammar is L-attributed iff:
    1. Each attr-eval. rule assigning a RHS inherited attr., args of rule; either inherited attr of LHS or appear in RHS to the left of attr. being assigned
    2. Each attr.-eval. rule assigning a LHS synth attr., args of rule: either inherited attr of the given LHS or attr. of some RHS symbol.
    3. Each attr-eval. rule assigning a synth attr. $_{action}$ of an action symbol, args inherited attr. of given action symbol.

* i.e. it can be parsed in one pass from left to right and bottom top top.

- **Simple Assignment Form iff:**
  1. Only non-copy rules are computing synth attrs of action symbols
  2. Set of copy rules for each production is independent.

(Can be achieved by inserting action symbol for function on args can be achieved through consolidation)

  - **Copy Rule:** Rule in the form $(x_1, x_2, \ldots, x_n) \leftarrow y$
  - Set of copy rules independent iff source (y) of each set does not appear in any other rules of the set.

- **Augmented Pushdown Machine:**
  - **Idea:** Given a simple-assignment-form, L-attributed grammar with an LL(1) input grammar, design a pushdown device which performs the described translation
    - First design pushdown processor ignoring attributes
    - Then augment its stack symbols with fields and augment transitions to compute attrs and fill in fields
    - When stack symbol is top of stack:
      1. **Inherited attrs:** field already contain value
      2. **Synth attrs:** field contains pointer to linked list of attr fields where attr to be stored
      3. **Input symbol attrs:** treated like synth attrs.
  - **Starting Configuration:** 

, field init to values specified in grammar (I) or null pointer (S)
  - **Input Symbol at Top:** Non-error case (b,b): attr values of input symbols copied to fields in list of fields pointed to by fields of top stack symbol, then POP, Advance

- Augmented Pushdown Machine:

  - Action Symbol at Top:
    1. (I-attrs retrieved from corresp fields of top stack symbol
    2. S-attrs computed from I-attrs using attr rules for Action Symbol
    3. Value of each S-attr place in each field in linked list of attrs pointed to by corresp field at top of stack
    4. POP, RETAIN

    *If string translation is being performed attrs also used to output a attributed output symbol.

  - Nonterminal at Top: Same as unaugmented but with attr. eval
    - Rule 1: each action symbol in production ($\langle A \rangle \to \xi b 4^a$) not pushed, synth attrs computed and attr outputs output from left to right
    - Rule 2: if copy rule "available" machine puts its value in certain fields of symbols on stack
    - Rule 3: if copy rule "not-available" → source attr of symbol to be pushed on stack, machine fills its field with ptr to list of fields to fill.

- Recursive Descent for Attributed Grammars: Same as before but with parameters:

  e.g. Procedure PROCS(R)
  Inherited attribute R.