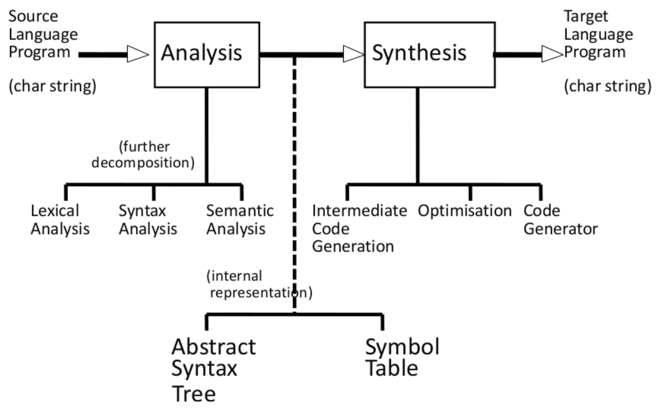
Revision Notes for CO221 Compilers

Autumn 2017



1 Lexical Analysis

Transforms a stream of characters into tokens.

1.1 Tokens

Identifier Tokens

- **Keyword identifiers**, e.g. `while` - generally have their own token.
- **Non-keyword identifiers** - have a general identifier token e.g. `IDENT("year")`.

Use a fast lookup function to determine if a scanned identifier is a keyword (e.g. perfect hash).

Literal Tokens

- **Unsigned integers** - represented by a token for integers, plus an integer.
- **Unsigned reals**, **strings** represented similarly.

Other Tokens

- **1 or 2-char symbols**, e.g. $+$, $<=$ usually represented by their own token.
- **Whitespace** and **comments** usually not represented in token stream.
- **Macros** usually removed before lexical analysis.

1.2 Regular Expressions

1. Write **test cases**.
2. Re-cast the question using standard characters instead of meta-characters.
3. Try drawing a DFA if easier.

	Matches
`a`	Symbol
ϵ	Empty string
`R1 R2`	`R1` followed by `R2`
`R1|R2`	`R1` or `R2`
`R*`	0 or more occurrence of `R`
`(R)`	`R`
`\a`	Escaped symbol
`R?`	0 or 1 occurrence of `R`
`R+`	1 or more occurrence of `R`
`[a-zA-Z]`	Any character from given set
`[^a-zA-Z]`	Any character except from given set
`.`	Any character except newline

1.3 Rules

Regular Expression Rules **Rules** (**productions**) of the form $\alpha \to$ `X` where α is a **non-terminal** (name of rule) and `X` is a regular expression constructed from both non-terminals and terminals (input chars).
Note that non-terminals must be defined before use, and recursion is not allowed.
E.g.

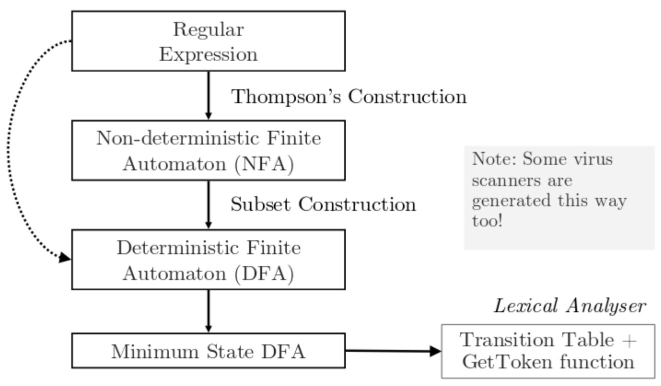
- `Digit` $\to$ `[0-9]`
- `Int` $\to$ `Digit+`

Disambiguation Rules

1. Use **longest matching character sequence**.
2. Assume regular expressions have **textual precedence** (earliest matching regular expression chosen).

Implementation Either hand-crafted, or use lexical analyser generator (e.g. Flex).

Lexical analyser generator:


Regular Expression
Thompson's Construction
Non-deterministic Finite Automaton (NFA)
Subset Construction
Deterministic Finite Automaton (DFA)
Minimum State DFA
Lexical Analyser
Transition Table + GetToken function
Note: Some virus scanners are generated this way too!

1.4 Finite Automata

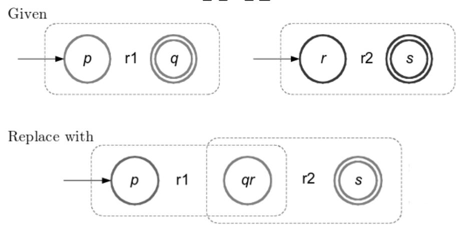
- **States** of matching process (circles).
- **Transitions** between those states (arrows).
- Matched input **symbols** (labels on transitions).
- **Accepting states** of matching process (double circle).
- **Start state** (unlabelled incoming arrow).

Deterministic Finite Automata No two transitions leaving a state have the same symbol.

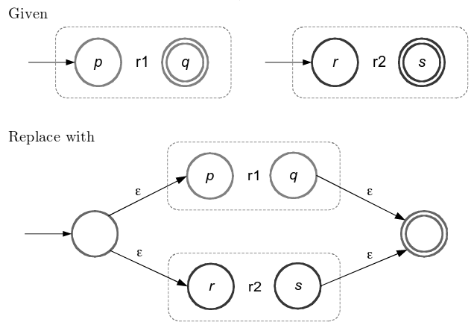
Non-deterministic Finite Automata Allow a choice of transitions out of a state

Thompson's Construction Use ϵ transitions to glue together automata for each part of a regex:

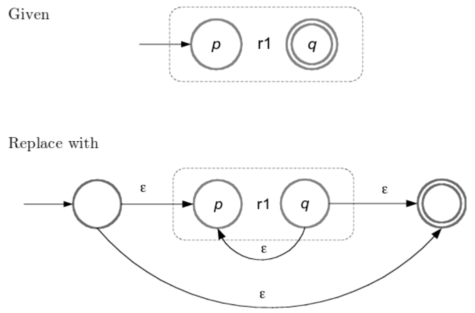
r1 r2



r1|r2



r1*



Subset Construction

- DFAs are much faster than NFAs (linear on size of input string),
- but require more memory (potentially 2^n states for an n-state NFA).

Algorithm:

1. DFA start state = ϵ closure of NFA start state.
2. For each new subset state S of the DFA:
   (a) For each unique symbol a leading out from any state of S:
      i. Add a transition a from S to S' where S' = ϵ closure of states reached by a

Generating a Lexical Analyser Encode DFA as a 2D table of states (row per state, col per char).

2 Parsing

Transforms a stream of tokens to an AST based on context free grammar of the language.

2.1 LR (Bottom-Up / Shift-Reduce) Parsing

- Can be implemented in $O(n)$ time.
- Several methods for generating LR parsing tables. States consist of LR(k) items of the grammar, vary in number of states and size of lookahead sets.

Context Free Grammars Test very carefully.

2.1.1 LR(0) Parsing

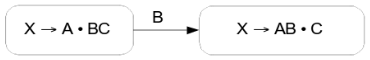
Don't use the current token in order to perform a reduce. E.g the rule $\texttt{X} \rightarrow \texttt{ABC}$ has 4 LR(0) items, where $\bullet$ indicates how much of the rule we have seen:

1. $\texttt{X} \rightarrow \bullet\texttt{ABC}$
2. $\texttt{X} \rightarrow \texttt{A}\bullet\texttt{BC}$
3. $\texttt{X} \rightarrow \texttt{AB}\bullet\texttt{C}$
4. $\texttt{X} \rightarrow \texttt{ABC}\bullet$

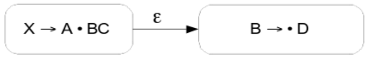
LR(0) to Finite Automata

1. Add a start rule with end-of-input symbol: e.g. $\texttt{E'} \rightarrow \texttt{E\$}$.
2. For each LR(0) item:

Given $\texttt{X} \rightarrow \texttt{A}\bullet\texttt{BC}$, we add the transition:



Suppose $\texttt{B}$ is a non-terminal, then for all $\texttt{B} \rightarrow \bullet\texttt{D}$, we add ϵ transitions of the form:



We then construct a DFA using subset construction.

1. Usually it's simple to construct the DFA directly.
2. Try and keep the DFA neat - otherwise it's easy to get lost.
3. Check very carefully for duplicate states.

Finite Automata to Parsing Table

1. For each terminal transition $\texttt{X} \xrightarrow{\texttt{T}} \texttt{Y}$, add $\texttt{P}[\texttt{X},\texttt{T}] = s\texttt{Y}$ (**shift** $\texttt{Y}$).
2. For each non-terminal transition $\texttt{X} \xrightarrow{\texttt{N}} \texttt{Y}$, add $\texttt{P}[\texttt{X},\texttt{N}] = g\texttt{Y}$ (**goto** $\texttt{Y}$).
3. For each state $\texttt{X}$ containing the item $\texttt{R'} \rightarrow \cdots\bullet$, add $\texttt{P}[\texttt{X},\$] = a$ (**accept**).
4. For each state $\texttt{X}$ containing the item $\texttt{R} \rightarrow \cdots\bullet$, add $\texttt{P}[\texttt{X},\texttt{T}] = rN$ (**reduce**), for every terminal $\texttt{T}$ where N is R's rule number.

Model of an LR Parser Modelled by pushdown automata:

- **Shift** pushes state onto stack and advances token.
- **Reduce**:
  - removes L tokens from the stack (L is the length of the RHS of the rule),
  - pushes the state of the **goto** for the LHS of the rule according to the state now on the top of the stack,
  - and generates an AST node for the rule.

2.1.2 Other LR Parsing Methods

- **FIRST set** for a sequence of non-terminals and terminals α is the set of all tokens that could start a derivation of α, including ϵ if α can derive ϵ.
  - A non-terminal $\texttt{A}$ is **nullable** if $A \implies {}^*\epsilon$.
- **FOLLOW set** for a non-terminal $\texttt{A}$ is the set of all tokens that could immediately follow $\texttt{A}$, including $ if $\texttt{A}$ can end the input. I.e. For each derivation of $\texttt{X} \rightarrow \texttt{ABC}$:
  1. `FOLLOW(B)` includes `FIRST(C)`.
  2. If `FIRST(C)` includes ϵ, then `FOLLOW(B)` includes `FOLLOW(X)`.
  3. If `C` is empty, then `FOLLOW(B)` includes `FOLLOW(X)`.

Check FOLLOW sets very carefully, and pay special attention to rules with ϵ in their FIRST set.

Different LR Parsers

- LR(0): A reduce item $\texttt{X} \rightarrow \texttt{A}\bullet$ always causes a reduction.
- SLR(1): A reduce item $\texttt{X} \rightarrow \texttt{A}\bullet$ causes a reduction only if the current token is in FOLLOW($\texttt{X}$).
- LR(1): A reduce item $\texttt{X} \rightarrow \texttt{A}\bullet,\texttt{t}$ causes a reduction only if the current token is $\texttt{t}$.
  - **FA Transitions**: For each rule $\texttt{X} \rightarrow \texttt{A}\bullet\texttt{BC},\texttt{t}$:

  - Add transition via $\mathtt{B}$ to $\mathtt{X} \to \mathtt{AB{\bullet}C,t}$.
  - For each rule $\mathtt{B} \to \mathtt{D}$ and every token $\mathtt{u}$ in $\mathtt{FIRST(Ct)}$, add transitions via ϵ to $\mathtt{B} \to \mathtt{D,u}$.

- LALR(1): Combines LR(1) states that differ in the look-ahead token only. This can cause spurious reductions before an error is detected.

2.1.3 Ambiguities

Grammars which can derive more than 1 parse tree for some input cannot be LR(k) for any k.

Shift-Reduce Conflicts Usually obvious in a DFA. E.g. Consider the grammar:

$$\begin{aligned} \mathtt{S} \to & \; if\ \mathtt{E}\ then\ \mathtt{S}\ else\ \mathtt{S} \\ \mathtt{S} \to & \; if\ \mathtt{E}\ then\ \mathtt{S} \\ \mathtt{S} \to & \; id \\ \mathtt{E} \to & \; 0 \mid 1 \end{aligned}$$

and the statement:

$$if\ \mathtt{a}\ then\ if\ \mathtt{b}\ then\ \mathtt{c}\ else\ \mathtt{d}$$

- If we shift first, we get: $if\ \mathtt{a}\ then\ \boxed{if\ \mathtt{b}\ then\ \mathtt{c}\ else\ \mathtt{d}}$.
- If we reduce first, we get: $if\ \mathtt{a}\ then\ \boxed{if\ \mathtt{b}\ then\ \mathtt{c}}\ else\ \mathtt{d}$.

Solution: rewrite the grammar:

$$\begin{aligned} \mathtt{S} \to & \; \mathtt{MatchedS} \\ \mathtt{MatchedS} \to & \; if\ \mathtt{E}\ then\ \mathtt{MatchedS}\ else\ \mathtt{MatchedS} \\ \mathtt{MatchedS} \to & \; id \end{aligned}$$

$$\begin{aligned} \mathtt{S} \to & \; \mathtt{UnmatchedS} \\ \mathtt{UnmatchedS} \to & \; if\ \mathtt{E}\ then\ \mathtt{S} \\ \mathtt{UnmatchedS} \to & \; if\ \mathtt{E}\ then\ \mathtt{MatchedS}\ else\ \mathtt{UnmatchedS} \end{aligned}$$

We now force $else$-parts to become matched as soon as possible.

Reduce-Reduce Conflicts Can usually tell from ambiguities. Consider the grammar:

$$\mathtt{Expr} \to \mathtt{Expr}\ \text{‘+’}\ \mathtt{Expr} \mid \mathtt{Expr}\ \text{‘*’}\ \mathtt{Expr} \mid \text{‘(’}\ \mathtt{Expr}\ \text{‘)’} \mid int$$

This is ambiguous because it doesn't define the associativity or precedence of + and *.
Solution: rewrite the grammar:

$$\begin{aligned} \mathtt{Expr} \to & \; \mathtt{Expr}\ \text{‘+’}\ \mathtt{Term} \mid \mathtt{Term} \\ \mathtt{Term} \to & \; \mathtt{Term}\ \text{‘*’}\ \mathtt{Factor} \mid \mathtt{Factor} \\ \mathtt{Factor} \to & \; \text{‘(’}\ \mathtt{Expr}\ \text{‘)’} \mid int \end{aligned}$$

Parse Trees

1. **Naranker**: build leaf nodes for each input token (**shift**) and non-leaf nodes for each rule (**reduce**).
2. **Paul**: one branch for every item on the RHS of a rule.

Abstract Syntax Trees Remove unnecessary information from the parse tree - can be constructed by a separate parse or by attaching AST construction code directly to grammar rules.

2.2 LL (Top-Down) Parsing

A language is LL(k) if k-token lookahead is sufficient to choose between alternatives of a rule when parsing.

Backus-Naur Form (BNF) The rules for a CFG take the form $\mathtt{A} \to \alpha$ where $\mathtt{A}$ is a rule and α is a possibly empty sequence of rules and tokens.

Extended BNF (EBNF)

- $\{\alpha\}$: 0 or more occurrences of α. (**Repetition**)
  - $\{\mathtt{E}\}$ becomes $\mathtt{X} \to \epsilon \mid \mathtt{XE}$
- $[\alpha]$: 0 or 1 occurrence of α. (**Optional**)
  - $[\mathtt{E}]$ becomes $\mathtt{X} \to \epsilon \mid \mathtt{E}$
- (α): α. (**Grouping**)

LL(1) A grammar is LL(1) if it:

1. For each distinct pair of alternatives (α, β) of a rule $\mathtt{A}$, FIRST(α) and FIRST(β) are disjoint.
2. For every rule $\mathtt{A}$, if FIRST($\mathtt{A}$) contains ϵ, then FIRST($\mathtt{A}$) and FOLLOW($\mathtt{A}$) are disjoint.

Recursive Descent Parser Consists of:

- Set of **parse functions**, one for each rule (which returns AST for the rule). E.g.

```
def BeginStatement():
    match(BEGIN)
    statlist = []
    statlist.append(Statement())
    while token == SEMICOLON:
        match(SEMICOLON)
        statlist.append(Statement())
    match(END)
    return BeginAST(statList)
```

- The current **input token**, e.g. as a variable.
- A token **match** and advance function.

```
def match():
    if token == expected:
        token = scanner.get_token()
    else error ("unexpected token...")
```

CFG to LL(1) 3 common transformations:

1. **Left recursion removal**: E.g.
   (a) A → X | AY is rewritten as A → X {Y}.
2. **Substitution**: Replace non-terminal A on LHS of a rule with each of the alternatives of A.
3. **Left factorisation**: E.g.
   (a) A → BC | BD is rewritten as A → B (C | D).
   (b) A → BC | B is rewritten as A → B [C].

Error Recovery We try to:

1. Produce informative messages,
2. If recovery is attempted, skip as little as possible,
3. If correction is attempted, ensure corrected program has same syntax tree and semantics as original.

In panic mode error recovery:

- Each parse function has a set of synchronising tokens (**syncset**). E.g.

```
def IfStatement(syncset):
    match(IF)
    Expr(syncset union {THEN, ELSE})
    match(THEN)
    Statement(syncset union {ELSE})
    check({ELSE}, syncset, "Error ...")
    if token  ==  ELSE:
        match(ELSE)
        Statement(syncset)
```

- When an error occurs, we skip ahead, discarding tokens until one of the synchronising tokens is seen.

```
def check(expectset, syncset, message):
    if token not in expectset:
        error(message)
        while token not in (expectset union syncset)
              and token != EOF:
            scanner .get_token() # skip tokens
```

3 Semantic Checking

We can check for errors during **parsing**, **semantic checking** (i.e. with a symbol table), and at **runtime**. Sometimes we **ignore** them (e.g. integer overflow, infinite loops).

Type Checking in Languages Can vary greatly. Languages may support:

- No explicitly typed variables (types inferred).
- Dynamically typed variables (types can change).
- Overloaded functions and polymorphic typing.
- Non-standard assignment-compatibility / type equivalence.
- Restrictions on how some types are used.
- Implicit type casting.
- Identifiers being used to name several kinds of entities.
- Multiple sizes for integers and floats.

Symbol Tables A dictionary from identifiers to AST nodes or objects holding identifier information. Standard implementation:

- **Top-level symbol table** pre-loaded with identifier entries for globally visible identifiers, e.g. standard types, constants, functions...
- New symbol table for each scope, with dictionaries from identifiers to types.

3.1 Example Checks

Variable Declaration

1. Check that the identifier is in scope.
2. Check type is valid, and objects of this type can be declared.
3. Check identifier isn't already defined.
4. Add variable to symbol table.

```
                            int Age

class VariableDeclAST:
    String typename                      # Syntactic attribute
    String varname                       # Syntactic attribute
    VARIABLE varObj                  # Semantic attribute

    def Check():                         # ST is the current Symbol Table
      T = ST.lookupAll(typename)
      V = ST.lookup(varname)
      if       T == None:         error("unknown type %s" % typename)
      elif ! T instanceof TYPE: error("%s is not a type" % typename)
      elif ! T.isDeclarable():  error("cannot declare %s objects" % typename)
      elif ! V == None:         error("%s is already declared" % varname)
      else varObj = new VARIABLE(T)      # varObj now holds a reference to T
           ST.add(varname, varObj)       # add to symbol table
```

Assignment

1. Check identifier is in scope and is a variable.
2. Check the types are compatible.

```
                              Age = E

class AssignmentAST:
    String varname                       # Syntactic attribute
    ExpressionAST expr                   # Syntactic attribute
    VARIABLE varObj                          # Semantic attribute

    def Check():
      V = ST.lookupAll(varname)
      expr.check()         # assume 'check' also records the type of expr
      if    V == None:     error("unknown variable %s" % varname)
      elif  ! V instanceof VARIABLE:
                           error("%s is not a variable" % varname)
      elif  ! assignCompat(V.type, expr.type):
                           error("lhs and rhs not type compatible")
      else  varobj = V
```

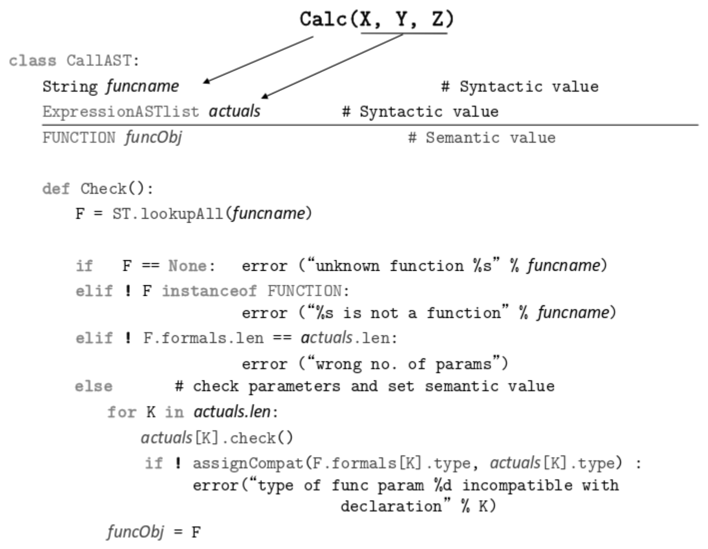
Function Declaration

1. Check return type is valid, and objects of this type can be returned.
2. Check the function isn't already defined.
3. Add function to the symbol table.
4. Create and link new symbol table.
5. Check parameter declarations.

```
class FunctionDeclarationAST:
    String returntypename;  String funcname;  ParameterASTlist parameters
    FUNCTION funcObj        # Semantic attribute

    def CheckFunctionNameAndReturnType():   # Similar to variable decl. check
        T = ST.lookupAll(returntypename)
        F = ST.lookup(funcname)
        if      T == None:   error ("unknown type %s" % returntypename)
        elif ! T instanceof TYPE:    error ("%s is not a type" % returntypename)
        elif ! T.isReturnable():
                             error ("cannot return %s objects" % returntypename)
        elif ! F == None:    error ("%s is already declared" % funcname)
        else
               funcObj = new FUNCTION(T)        # link to T and parameter list
               ST.add(funcname, funcObj)     # add F to symbol table

    def Check():
      CheckFunctionNameAndReturnType():

      ST = new SymbolTable(ST)      # create and link new symbol table
      funcObj.symtab = ST

      for P in parameters:          # check parameter declarations
          P.check()
     funcObj.formals.append(P.paramObj)

      ST = ST.encSymTable           # return to enclosing symbol table
```

Function Call

1. Check function exists.
2. Check length of parameters.
3. Check parameters.
4. Check return type.



```
                          Calc(X, Y, Z)
class CallAST:
    String funcname                              # Syntactic value
    ExpressionASTlist actuals           # Syntactic value
    ---------------------------------------------------------------
    FUNCTION funcObj                          # Semantic value

    def Check():
       F = ST.lookupAll(funcname)

       if   F == None:   error (“unknown function %s” % funcname)
       elif ! F instanceof FUNCTION:
                         error (“%s is not a function” % funcname)
       elif ! F.formals.len == actuals.len:
                         error (“wrong no. of params”)
       else      # check parameters and set semantic value
          for K in actuals.len:
             actuals[K].check()
             if ! assignCompat(F.formals[K].type, actuals[K].type) :
                  error(“type of func param %d incompatible with
                                   declaration” % K)
          funcObj = F
```

4 Runtime Memory Organisation

Basic Types

- **Primitive types**: mapped to memory locations and/or registers. Sometimes memory aligned to keep accesses optimal.

- **Records / structures**: fields are grouped together and usually allocated consecutively in memory. Fields accessed by a constant byte offset from start.

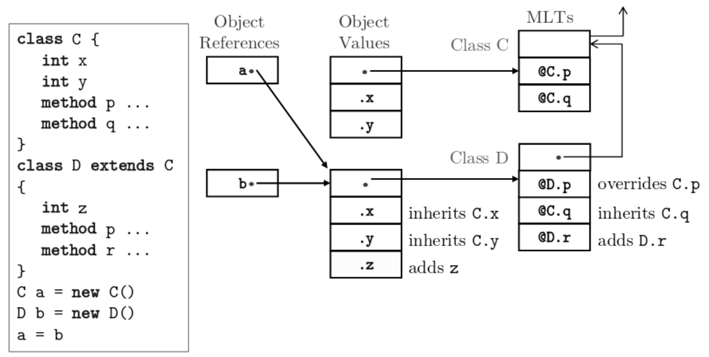
- **Arrays**: variables of same size grouped together and accessed by a runtime integer expression that gives the offset from the start.

Objects

- **Fields**: memory location of object reference + offset of field.

- **Methods**: object reference passed as a hidden parameter.

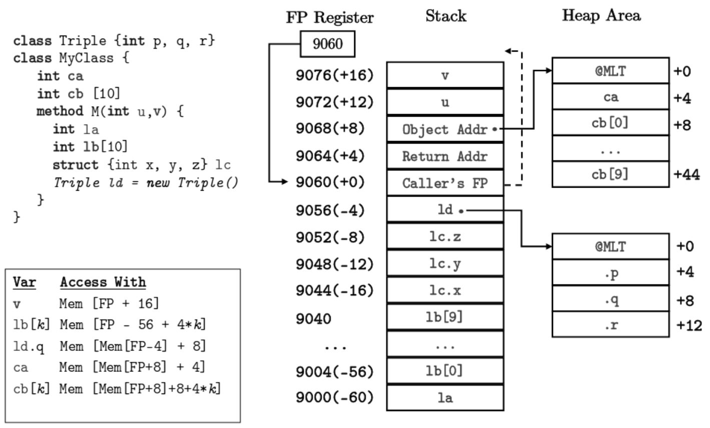
- **Inheritance**, **overriding** and **dynamic binding**:



```
class C {
  int x
  int y
  method p ...
  method q ...
}
class D extends C
{
  int z
  method p ...
  method r ...
}
C a = new C()
D b = new D()
a = b
```

Program Address Space

- **Code segment**.
- **Stack segment**: for storing local variables.
  - Access is through the **frame pointer register**.
  - For **method calls**: we also hold the return address, address of the method's of object, parameters, saved values of registers, space for a return value.



```
class Triple {int p, q, r}
class MyClass {
   int ca
   int cb [10]
   method M(int u,v) {
     int la
     int lb[10]
     struct {int x, y, z} lc
     Triple ld = new Triple()
   }
}
```

Var	Access With
v	Mem [FP + 16]
lb[k]	Mem [FP - 56 + 4*k]
ld.q	Mem [Mem[FP-4] + 8]
ca	Mem [Mem[FP+8] + 4]
cb[k]	Mem [Mem[FP+8]+8+4*k]

- **Data segment**.
  - **Static area**: for storing global / static variables (also constants and MLTs sometimes).
  - **Heap area**.
    - ∗ Keep a **FreeList** (chain of pointers to unused / returned / garbage collected memory blocks in the heap area).
    - ∗ Maintain different free lists for different block-size intervals.

Garbage Collection The garbage collector needs to know:

- Which variables in the program point to the heap.
- For each object pointed to, which pointers it contains.

Heap compaction: Reduce fragmentation by:

1. Mark live blocks.
2. Co-locate live blocks.
3. Update the pointers that point to co-located blocks.

Algorithms include:

1. **Reference-Counting.** Record in each block the number of pointers that point to the block.
   - Needs techniques to reclaim cyclic data structures.
2. **Mark-Sweep.** Has two phases:
   (a) **Mark phase**: mark all blocks that are reachable from non-heap references.
   (b) **Sweep phase**: scan all blocks, reclaiming dead blocks and unmarking live blocks.
   - **Pointer-Reversal Marking**: allows visiting all notes of a directed graph without additional stack space. Swap children pointer with parent pointer.
3. **Two-Space.** Split the heap into a From-Space and a To-Space.
   (a) Allocate blocks from the From-Space.
   (b) When the From-Space is exhausted, copy reachable blocks from the From-Space to the two space.
   - **Fast**: no pointers manipulations, copying automatically compacts the heap.
   - **Wasteful**: wastes half the memory and copies long-lived blocks.
4. **Generational.** Divide the heap into several generations based on age of block.
   - Allocate new blocks from the youngest generation.
   - Perform GC on youngest generation more often than the next.
   - Apply different GC techniques to different generations.

Debuggers Reverse lookup from PC to source line.

- Don't affect behaviour of a program except execution time.
- Requires information about identifiers and mapping from program counter to source lines.

Profilers Interrupt program every X milliseconds.

- Identify the method that was executing - lookup table sorted by start address of methods.
- Increment a counter.
- Print out counts on program completion.

5 Code Generation

For each statement / expression type, use a standard template, with gaps filled in with details.

Stack Machine Uses a stack and a single temporary register. E.g. `Add: T := store[SP]; SP := SP+1; T := store[SP]+T; store[SP] := T`.

```
data Instruction
  = Add | Sub | Mul | Div ...
  | PushImm Int     -- push constant onto stack
  | PushAbs Name    -- push variable at given Name to stack
  | Pop Name        -- remove top of stack and store at Name
  | CompEq          -- subtract top two elements of stack,
                    -- replace with 1 if 0, 0 otherwise
  | JTrue Label     -- remove top of stack, jump if 1
  | JFalse Label    -- jump if 0
  | Define Label    -- set up destination for jump

transStat :: Stat -> [Instruction]
transStat (Assign id exp)
  = transExp exp ++ [Pop id]
transStat (Seq s1 s2)
  = transStat s1 ++ transStat s2
transStat (ForLoop id e1 e2 body)
  = transExp e1 ++ [Pop id] ++
    [Define start] ++
    transExp e2 ++ [PushAbs id] ++ [CompGt] ++
    [JTrue end] ++
    tranStat body ++
    [PushAbs id] ++ [PushImm 1] ++ [Add] + [Pop id] ++
    [Jump start] ++
    [Define end]

transExp :: Exp -> [Instruction]
transExp (Binop op e1 e2)
  = transExp e1 ++
    transExp e2 ++
    transBinop op
transExp (Ident id)
  = [PushAbs id]
transExp (Const v)
  = [PushImm v]

transBinop :: Op -> [Instruction]
transOp Plus = [Add]
transOp Minus = [Sub]
transOp Times = [Mul]
transOp Divide = [Div]
```

Register Machine Assumes an infinite number of registers. E.g. `Add R1 R2`: `R1 := R1 + R2`.

```
data Instruction
  = Add Reg Reg | Sub Reg Reg | ...
  | Load Reg Name      -- Reg := value at locn Name
  | LoadImm Reg Int    -- Load constant into Reg
  | Store Reg Name     -- Store Reg at locn Name
  | Push Reg           -- Push Reg onto stack
  | Pop Reg            -- Remove val from stack and put in Reg
  | CompEq Reg Reg
    | JTrue Reg Label | JFalse Reg Label | Define Label

transExp :: Exp -> Reg -> [Instruction]
transExp (Binop op e1 e2) r
  = transExp e1 r ++
    transExp e2 (r+1) ++
    transBinop op r (r+1)
transExp (Ident id) r
  = [Load r id]
transExp (Const n) r
  = [LoadImm r n]

transBinop :: Op -> Reg -> Reg -> [Instruction]
transBinop Plus r1 r2 = [Add r1 r2]
```

- Can be improved by using immediate operands instead of always loading constants into registers.

Accumulator Machine Use one register. E.g. `Add`: `Acc := Acc+Store[SP]; SP := SP+1`.

```
data Instruction
  = Add | Sub | Mul | Div ...
  | Push           -- SP := SP-1; Store[SP] := Acc
  | Pop            -- Acc := Store[SP]; SP := SP+1
  | Load Name      -- Acc := Store[Name]
  | LoadImm Int    -- Acc := Int
  | Store Name     -- Store[Name] := Acc
  | Jump Label | JTrue Label | JFalse Label | Define Label

transExp :: Exp -> [Instruction]
transExp (Binop op e1 e2)
  = transExp e2 ++
    [Push] ++
    transExp e1 ++
    transOp op
transExp (Ident id)
  = [Load id]
transExp (Const n)
  = [LoadImm n]
```

Machines with Limited Register Sets

- While free registers remain, use the register machine strategy.
- When the limit is reached, revert to the accumulator strategy.

```
transExp :: Exp -> [Instruction]
transExp (Binop op e1 e2)
  | r == MAXREG = transExp e2 r ++
                  [Push r] ++
                  transExp e1 r ++
                  transBinopStack op r
  | otherwise   = transExp e1 r ++
                  transExp e2 (r+1) ++
                  transBinop op r (r+1)
transExp (Ident id) r
  = [Load r id]
transExp (Const n) r
  = [LoadImm r n]

transBinop :: Op -> Reg -> Reg -> [Instruction]
transBinop Plus r1 r2 = [Add r1 r2]

transBinopStack :: Op -> Reg -> [Instruction]
transBinopStack Plus r = [AddStack r]
```

5.1 Register Allocation

Calling a Function Two options:

- **Caller saves registers**: but doesn't know which registers the callee needs - has to save all registers it's using that might be used.
- **Callee saves registers**: but doesn't know which registers the callers need - has to save all registers that it needs that might be being used.

Don't forget to:

1. Keep the **parameter register**(s) **safe**.
2. **Restore** the registers (in **reverse** order).

Sethi-Ullman Numbering Weight the syntax tree.

- Order of evaluation of subexpressions is important: evaluate the one that uses the **most** registers **first**.
  - **Register targeting**: we want to leave the result in register R, but also not use the register $R+1$. Pass a **list** of registers its allowed to use and return the result in the first one.
  - **Functions**: often best to do first, since we may have to save other registers.

```
weight :: Exp -> Int
weight (Const n) = 1
weight (Ident id) = 1   -- Assuming loads into register
weight (Binop op e1 e2)
  = min [cost1, cost2]
  where
    cost1 = max[weight e1, weight e2 + 1]
    cost2 = max[weight e1 + 1, weight e2]

transExp :: Exp -> [Reg] -> [Instruction]
transExp (Const n) (dstreg:regs)
  = [LoadImm dstreg n]
transExp (Ident id) (dstreg:regs)
  = [LoadAbs dstreg id]
transExp (Binop op e1 e2) (dstreg:nxtreg:regs)
  = if weight e1 > weight e2 then
    transExp e1 (dstreg:nxtreg:regs) ++
    transExp e2 (nxtreg:regs) ++
    transBinop op dstreg nxtreg
  else
    transExp e2 (nxtreg:dstreg:regs) ++
    transExp e1 (dstreg:regs) ++
    transBinop op dstreg nxtreg
```

- **Worst case**: a perfectly-balanced expression tree with k operators and $k-1$ intermediate values. $\lceil \lg k \rceil$ registers required.

- Fails to exploit context:
  - Doesn't use registers from statement to statement, e.g. using variables.

Graph Colouring Allocation

1. Use a tree-walking translator to generate an **intermediate code** that saves temp values in named locations (three-address code).
2. Construct the **interference graph**. Each pair of nodes is linked if the values must be stored simultaneously (**live ranges overlap**).
3. Attempt to colour the nodes so that no connected nodes have the same colour.

- We can now make use of more advanced optimisations such as saving **common subexpressions**.
- **Spilling**: if we fail to colour the graph, choose a variable whose live range is causing trouble, and split its range by storing it to memory and reloading it later.
  - Avoid spilling in innermost loop.

6 Optimisation

- **High level**: e.g. **inlining**: replace a call `f(x)` with the function body itself.
- **Low level**: e.g. **instruction scheduling**: reorder instructions so processor can execute them in parallel.

Peephole Optimisation Scan assembly code, replacing obviously inane combinations of instructions.

Constant Propagation Evaluate values at compile time instead of at runtime.

Dead Code Elimination Find code that's never needed and remove it.

Spectrum of Possible Optimisations

- **Local**: Optimisation works at level of basic blocks. E.g. Sethi-Ullman algorithm, peephole optimisation. Runs quickly and easy to validate.
- **Global**: Optimisation works on a whole procedure. May have worse than linear complexity.
- **Interprocedural**: Works on the whole program. Hard to do.

Loop Optimisations

- **Common subexpressions**.
- **Loop invariant code motion**: instructions whose operands only arrive from outside the loop.
  - Move loop-invariant instructions into loop header.
- Detection of **induction variables**: variables which increase / decrease by a constant each iteration.
- **Strength reduction**: calculate induction variable by incrementing instead of multiplying by other induction variables.
- **Control variable selection**: replace control variable with induction variable.
- **Loop rotation**: to get rid of an unconditional jump.
- **Loop unrolling**.

Intermediate Representations

- Represent primitive operations necessary to execute program.
- Uniform representation, easy to analyse and manipulate.
- Independent of target instruction set.

Multiple levels of IR is popular, e.g.

1. **Tree**: before instruction selection.
2. **Flow graph**: after instruction selection.

Assign registers only after optimisation is complete. Until this point, use **temporaries**.

6.1 Dataflow Analysis

A Control Flow Graph defines:

1. List of temporaries which this instruction **updates**.
2. List of temporaries which this instruction **reads**.
3. List of nodes which may be **next**.

The algorithm:

1. Sets up simultaneous equations by writing properties in terms of **nodes** or **points** (edges in and out of each node) in a CFG.
2. Solves simultaneous equations by iteration. Consider:
   - **Does it terminate?** What's the maximum / minimum size of the sets? Can sets loose / gain members?

Example 1: **Live Variable Analysis.**

- `x` is live at a point p (between adjacent nodes) if the value of `x` can be used along some path starting at p.
  - A variable is live **after** node n if it is live before any of n's successors.

$$\text{LiveOut}\,(n) = \bigcup_{s \in \text{succ}(n)} \text{LiveIn}\,(s)$$

  - A variable is live **before** node n if it is:
    1. Used by node n, or
    2. Alive after node n and not overwritten by node n.

$$\text{LiveIn}\,(n) = \text{uses}\,(n) \cup (\text{LiveOut}\,(s) - \text{defs}\,(n))$$

- We end up with a system of simultaneous equations. Solve by iteration:
  1. Start with `LiveIn` and `LiveOut` $= \{\}$ for each node.
  2. Iterate over each each node in graph (**backwards**), and update until they don't change any more.
- Find all interferences. For each temporary:
  - Iterate through all points and add any interfering temporaries.
- Use graph colouring to allocate registers.

Example 2: **Loop-Invariant Code Motion.**

1. **Reaching Definitions.**
   - A **definition** of `x` is a statement which may assign to `x`.
   - A definition d **reaches** a point p if there exists a path from d to p s.t. d is not killed along that path.
     - The Gen (n) set is the set of definitions generated by a node n (usually $\{n\}$, but $\{\}$ for branches).
     - The Kill (n) set is the set of all definitions of the variable except n.
     - A variable reaches **before** a node n, if it reached any of n's predecessors.

$$\text{ReachIn}\,(n) = \bigcup_{p \in \text{pred}(n)} \text{ReachOut}\,(p)$$

     - A variable reaches **after** a node n, if it was:

(a) Generated by n, or
(b) Reached before n and is not killed by n.

$$\text{ReachOut}(n) = \text{Gen}(n) \cup (\text{ReachIn}(n) - \text{Kill}(n))$$

- This time we iterate **forwards**.
- A definition is loop-invariant if for each $u_i$that d uses:

(a) The definitions of u_i that reach d are outside the loop, or
(b) Only one definition of u_i reaches d, and that definition is loop invariant.

2. **Loops.**

- A **loop** (in a CFG) is a set of nodes S including a header node h such that:

(a) From any node in S there is a path to h.
(b) There is a path from h to any node in S.
(c) There is no edge from any node outside S to any node in S other than h.

(a) **Dominators.**

- Node d dominates n if every path from the start node to n goes through d.
  - The start node s is dominated by itself.
  - Node n is dominated by any node that dominates **all** of its predecessors.

$$\text{Doms}(n) = \begin{cases} \{n\} & n \text{ is the start node} \\ \{n\} \cup \left(\bigcap_{p \in \text{preds}(n)} \text{Doms}(p)\right) & \text{otherwise} \end{cases}$$

- Iterate forwards **on nodes** (not points). This start with all nodes and make sets smaller.

(b) **Natural Loops.**

- A **back edge** is an edge from a node n to a node h that dominates n.
  - The **natural loop** of a back edge (n, h) is the set of nodes x such that:
    i. h dominates x, and
    ii. There is a path from x to n not containing h.
  - h is the **header** of this loop.
  - If the natural loop contains a header for another loop, then it is a **nested loop**.
- Move loop-invariant **instructions** to a **pre-header** node above the header node.

- An instruction is loop invariant and can be hoisted if:

(a) **Definition is loop-invariant**: use reaching definitions DFA.
(b) **Loop invariant node dominates all loop exits**: use dominators DFA.
(c) **Only a single definition of this variable in loop**: simple count.
(d) **Defined variable must not be live after loop's pre-header**: use live variables DFA.