Linear codes

- Alternative to syntax tree/graph representations of code
- Select a set of abstract operations that cover all the desired functionality in the languages targetted
- Each abstract operation specifies a fixed number of operands and a location to store the results, e.g.
- Copy operation x = y
- Addition operation x = y + z
- Rewrite the parsed source code using the abstract operations

One-address codes

- One approach is to use single-address codes
- These provide an operation and possibly one operand
- All other operands are assumed to be on top of a stack (with pre-defined expected order)
- All results are pushed onto the stack

Example: one address

Modelling x = 3 + y * z

- push z
- push y
- mult (pops z,y, performs *, pushes result)
- push 3
- add (pops 3, result of y*z, performs +, pushes result)
- pop x (pops result off stack into x)

Three address codes

- Assume binary operations, each with form

  ITEM1 = ITEM2 OP ITEM3

- Good for modelling target assembly languages
- Often assumes items are literals or internal variables
- Memory operations assumed to be expensive, so program variables often copied to/from internal variables before and after use, with some form of copy operator, e.g.

  ITEM1 = ITEM2

- May augment with other unary operations

Three-address example

- x = a + b * a + x
  - R1 = x // temp vars for each of the program variables
  - R2 = a
  - R3 = b
  - R4 = R3 * R2 // temp vars for each internal result
  - R5 = R2 + R4
  - R6 = R5 + R1
  - X = R6 // copy result back to program variables at end

Need to support variety of ops

- Three-address operations need to support variety of operations, jumps, conditional checks, array indexing, etc
- Need to come up with a suitable suite of 3-address ops
- Examples, using C-like syntax

- x = y OP z
- x = OP y
- x = y
- if x goto L
- ifnot x goto L

- x = y[z]
- x[y] = z
- x = &y
- x = *y
- *x = y

Function calls push params 1st:

```
param x1
param x2
...
param xn
Y = call p, n
```

Implementation choices

- Could implement simply as an array of records, using a value-number approach
- Each temp/internal variable gets one array position, so we can refer to the temp variable simply by index position
- Record specifies optype, temp var destination, arg1, arg2
- Choosing array size: too big and we're wasting space, too small means we'll need to resize somehow later
- Complicates re-ordering statements later: need to find and update all the instructions that refer to that index

Pointer based approach

- Could make array of pointers to records, and replace index references with pointers to correct records
- Makes rearranging order of array elements trivial
- Still has the problem of choosing an appropriate array size (though now if the array is too big we're simply storing extra pointer space, not entire extra unused records)

ADT approaches

- Could use linked list of records instead of an array
  - Eliminates too-big/too-small array issue
  - Introduces complications in searching the set of records (has to be a linear search)
- Could use hash-table approach, with each bucket containing a linked list of records
  - Need a good hash function, but effective searching if bucket sizes are small
  - Still need a way to thread the statement ordering

Single static assignment SSA

- Variant in which each variable is assigned to only once
- Introduces new intermediate variables as needed
- Variables used thus less tied to original source code, can provide better flexibility for optimizations later

Original	SSA
x=i*j	x1=i*j
y=k-x	y1=k-x1
x=y+a	x2=y1+a
y=x+i	y2=x2+i
y=y+a	y3=y2+a