Phases of a compiler

Intermediate Representation (IR): specification and generation

Figure 1.6, page 5 of text
Variables and memory

- Variables have names in our high level programs
- Names don't exist at runtime
- Variables are allocated space in a block of memory
  - Local variables have space in a stack frame (a.k.a. invocation record)
  - Array cells and record members have space in heap-allocated block of member
Variables and memory

- Every use of a variable is translated into an address by the compiler...
  
  - but not an absolute address!

- For every allocated block of memory there is a base/reference address.

- Variables housed within each block has a location in the block that is relative to the base/reference address.
Variables and memory

- The relative address is expressed as an **offset** from the base/reference address.
- The offset is determined by:
  - where other variables in the block are located,
  - how much space is needed to hold the variable's type of value, and
  - whether or not we need to align the starting address on a specific boundary.
What is the size of a multi-dimensional array of type T?

sizes of dimensions ($S_i$): $X \times 4$ bytes

data: $(\prod_{i \in X} S_i) \times \text{sizeof}(T)$

address for size of second dimension (and offset 4)

address for $a(0,0)$: offset 8

address for $a(0,1)$: offset 9

address for $a(0,2)$: offset 10

address for $a(1,0)$: offset 11

e tc.

size of first dimension

3

size of second dimension

a(0,0)
a(0,1)
a(0,2)
a(1,0)
a(1,1)
a(1,2)

first row

second row
Scopes

records (in separate symbol table), sequence of declarations at start of sblock

dblocks (6.3.5 and 6.3.6)

Since declarations must be gathered together at the start of an sblock, and cannot themselves be directly nested: keep running offset, but remember old offset when entering embedded scope.

dblock \rightarrow \['
    { Stack.push(offset); }
    declaration-list '
    
    { offset=Stack.pop(); }

\[
\begin{array}{llll}
\text{offset} & 0 & 4 & 8 \\
\text{integer}: x & 16 & 24 & \\
\text{integer}: y & & & \\
\text{real}: x & & & \\
\text{real}: z & & & \\
\text{integer}: x & 8 & 8 & \\
\text{integer}: y & & & \\
\text{real}: x & & & \\
\text{real}: z & & & \\
\text{Boolean}: y & & & \\
\text{character}: z & & & \\
\end{array}
\]
Dealing with alignment

"On many machines, instructions [...] may expect integers to be aligned, that is, placed at an address divisible by 4" [p. 428]
Dealing with alignment

{
  [ Boolean : a ; integer : x ; character c ; real : y ]
}

{
  [ character : d ; integer : r , s ] ... }

{
  [ Boolean : f , g ; real : t ; character h ] ... }

"On many machines, instructions [...] may expect integers to be aligned, that is, placed at an address divisible by 4" [p. 428]
Dealing with alignment

"On many machines, instructions [...] may expect integers to be aligned, that is, placed at an address divisible by 4" [p. 428]
Dealing with alignment

{ [ Boolean : a ; integer : x ; character c ; real : y ]
{ [ character : d ; integer : r, s ] ... }  
{ [ Boolean : f, g ; real : t ; character h ] ... }
}

"On many machines, instructions [...] may expect integers to be aligned, that is, placed at an address divisible by 4" [p. 428]
Dealing with alignment

Blocks are aligned, but memory wasted to padding
Dealing with alignment

{ [ Boolean : a ; integer : x ; character c ; real : y ]
{ [ character : d ; integer : r , s ] ... }
{ [ Boolean : f , g ; real : t ; character h ] ... }
}

Blocks are aligned, no padding needed here.
Dealing with alignment

```
{ [ Boolean : a ; integer : x ; character c ; real : y ]

[ character : d ; integer : r , s ] ... }

{ [ Boolean : f , g ; real : t ; character h ] ... }
}
```

Blocks are aligned, padding needed before embedded scope block.
Three address code instructions

- Now we start in earnest to think about generating (intermediate) code for various expressions and statements.
- Any instruction can have a label.
- Labels are the targets of jump instructions.
Three address code instructions
(see 6.2.1, pages 364-5)

1. \( x = y \ \text{op} \ z \)
2. \( x = \text{op} \ y \)  
   (treat i2r and r2i as unary ops)
3. \( x = y \)
4. goto L
5. if x goto L / ifFalse x goto L
6. if x \( \text{relop} \) y goto L
7. function calls:
   - param x
   - call p, n
   - y = call p
   - return y
8. \( x = y[i] \) and \( x[i] = y \)
9. \( x = \&y, x = *y, *x = y \)

Need operators for various types: let's use \(<t>\text{op}, \text{as in i+ or r* or b<}\)
Three address code instructions
(see 6.2.1, pages 364-5)

1. \( x = y \ op \ z \)
2. \( x = \ op \ y \)  \hspace{1cm} \text{(treat i2r and r2i as unary ops)}
3. \( x = y \)
4. \( \text{goto L} \)
5. if \( x \) goto L / ifFalse \( x \) goto L
6. if \( x \) relop \( y \) goto L
7. function calls:
   - param \( x \)
   - call \( p, n \)
   - \( y = \text{call} \ p \)
   - return \( y \)
8. \( x = y[i] \) and \( x[i] = y \)
9. \( x = &y, \ x = *y, \ *) x = y \)

Need operators for various types: let's use <t>op, as in i+ or r* or b<
Skip 6.5.4-6.5.6
Control flow

- Booleans to control flow
- Booleans as values
Boolean expressions

Examples: \(! X \quad X \& Y \quad X \mid Y\)
Boolean expressions

Examples:  ! X  X & Y  X | Y

We will do short-circuit evaluation
Boolean expressions

- Examples: \(! X\) \ X \ & Y \ X \mid Y
- We will do short-circuit evaluation
- if \((X \mid Y \& Z)\) then \{ A \} else \{ B \} is translated as

```
if X goto LA
ifFalse Y goto LB
ifFalse Z goto LB
LA:   A
      goto END
LB:   B
END: (next instruction)
```