Phases of a compiler

Intermediate Representation (IR): specification and generation

Figure 1.6, page 5 of text
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.
**dblocks (6.3.5 and 6.3.6)**

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

Since declarations must be gathered together at the start of an sblock, and cannot themselves be directly nested, we can do better:

```
dblock -> '['
  {  Env.push(st); st = new Env(); Stack.push(offset); offset = 0;  }
  declaration-list ']'
  {  dblock.type=record(st); dblock.width=offset; st=Env.pop(); offset=Stack.pop();  }
```

Since declarations must be gathered together at the start of an sblock, and cannot themselves be directly nested, we can do better:

```
offset = 0
offset = 4
offset = 8
push offset = 8 onto stack

offset = 8
offset = 16
offset = 24
offset = 8
pop offset = 8 from stack
push offset = 8 onto stack

offset = 8
offset = 8
offset = 9
offset = 10
push offset = 8 onto stack

offset = 8
offset = 8
offset = 9
offset = 10
offset = 11
offset = 12
offset = 13
offset = 14
push offset = 8 onto stack

offset = 8
offset = 8
offset = 9
offset = 10
push offset = 8 onto stack

offset = 8
offset = 8
offset = 9
offset = 10
offset = 11
offset = 12
offset = 13
offset = 14
push offset = 8 onto stack

offset = 8
offset = 8
offset = 9
offset = 10
push offset = 8 onto stack

offset = 8
offset = 8
offset = 9
offset = 10
offset = 11
offset = 12
offset = 13
offset = 14
push offset = 8 onto stack

offset = 8
offset = 8
offset = 9
offset = 10
```

**AT RUNTIME**
**dblocks (6.3.5 and 6.3.6)**

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

Since declarations must be gathered together at the start of an sblock, and cannot themselves be directly nested, we can do better:

\[
dblock \rightarrow '[$
\quad \{ \Env.push(st); \ st = \new \Env(); \ Stack.push(offset); \ offset = 0; \} \ '
\quad \text{declaration-list '}$
\quad \{ \dblock.type=\record(st); \ dblock.width=offset; \ st=\Env.pop(); \ offset=\Stack.pop(); \} 
\]

\[
\begin{align*}
\text{offset} & = 0 \\
\{ (\text{integer} : x, y) \} & \quad \text{push offset = 8 onto stack} \\
\{ (\text{real} : x, z) \ldots \ldots \} & \quad \text{offset = 8} \\
\{ \text{offset = 16} \\
\{ \text{offset = 24} \\
\{ (\text{Boolean} : y; \text{character} : z) \ldots \ldots \} & \quad \text{pop offset = 8 from stack} \\
\{ \text{offset = 9} \\
\{ \text{offset = 10} \\
\{ \ldots \\
\{ \ldots \\
\} & \quad \text{push offset = 8 onto stack} \\
\} & \quad \text{offset = 8} \\
\} & \quad \text{offset = 8} \\
\} & \quad \text{offset = 8} \\
\} & \quad \text{offset = 8} \\
\} & \quad \text{offset = 8} \\
\} & \quad \text{offset = 8} \\
\} & \quad \text{AT RUNTIME}
\end{align*}
\]
dblocks (6.3.5 and 6.3.6)
records (in separate symbol table), sequence of declarations at start of sblock

Since declarations must be gathered
together at the start of an sblock, and cannot themselves be directly nested,
we can do better:

\[
\text{dblock} \rightarrow [' \{ \text{Env.push(st); st = new Env(); Stack.push(offset); offset = 0; } \}
\text{declaration-list ']} \}
\text{dblock.type=record(st); dblock.width=offset; st=Env.pop(); offset=Stack.pop(); }
\]

\[
\begin{aligned}
&\text{offset = 0} \\
&\text{offset = 4} \\
&\text{offset = 8} \\
&\text{push offset = 8 onto stack} \\
&\text{offset = 8} \\
&\text{offset = 16} \\
&\text{offset = 24} \\
&\text{pop offset = 8 from stack} \\
&\text{push offset = 8 onto stack} \\
&\text{offset = 8} \\
&\text{offset = 9} \\
&\text{offset = 10} \\
&\text{push offset = 8 onto stack} \\
&\text{offset = 8} \\
&\text{offset = 16} \\
&\text{offset = 24} \\
&\text{pop offset = 8 from stack} \\
&\text{} \end{aligned}
\]

\(\text{AT RUNTIME}\)
dblocks (6.3.5 and 6.3.6)
records (in separate symbol table), sequence of declarations at start of sblock

Since declarations must be gathered together at the start of an sblock, and cannot themselves be directly nested, we can do better:

dblock → '[
  {  declaration-list ']
  {  dblock.type=record(st); dblock.width=offset; st=Env.pop(); offset=Stack.pop(); }
**dblocks (6.3.5 and 6.3.6)**

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

Since declarations must be gathered together at the start of an sblock, and cannot themselves be directly nested, we can do better:

```plaintext
dblock → '[:
  declaration-list ']
  { dblock.type=record(st); dblock.width=offset; st=Env.pop(); offset=Stack.pop(); }
```

Since declarations must be gathered together at the start of an sblock, and cannot themselves be directly nested, we can do better:

```plaintext
{ integer : x , y }
push offset = 8 onto stack
{ real : x , z } ...
offset = 16
offset = 24
pop offset = 8 from stack
push offset = 8 onto stack
{ boolean : y ; character : z }
offset = 9
offset = 10
pop offset = 8 from stack
}
```

**AT RUNTIME**
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 ] ... } }
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

{ [ 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] 

A block of size N bytes typically needs be aligned to an address divisible by N, where N is an integral power of 2 (1, 2, 4, 8) 

Blocks are not aligned.
Dealing with alignment

Blocks are aligned, but memory wasted to padding.

C will lay fields out in the order listed in the struct declaration.
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.
Offsets and alignment in the project

- The offsets for each variable in a scope is stored in its symbol table entry.

- The offsets must respect alignment constraints.
  - assume real is aligned to an 8-byte address boundary
  - assume int is aligned to a 4-byte boundary
  - assume smaller types can be at any address
  - assume reserve returns an address on an 8-byte boundary
Offsets and alignment in the project

- You may align using padding alone.
- You may align using a combination of re-organization of fields (large blocks before small blocks) and padding as necessary.
- Compute offsets during processing, and record in symbol table.
IR: a motivating example
Flow-of-Control (6.3.3)

if ( B ) then S1 else S2

Let’s generalize from the previous concrete example to one with an arbitrary Boolean expression B.

We assume that IR instructions are placed into an array.
**Flow-of-Control (6.3.3)**

if ( B ) then S1 else S2

B.true = newlabel()
B.false = newlabel()
S.next = S1.next = S2.next
S.code = B.code ||
label(B.true) || S1.code ||
gen('goto' S.next) ||
label(B.false) || S2.code
Flow-of-Control (6.3.3)

S -> if ( B ) then S1

B.true = newlabel()
B.false = S.next = S1.next
S.code = B.code || label(B.true) || S1.code

ifTrue:
goto LS1
ifFalse:
goto END
while ( B ) then S1

begin = newlabel()
B.true = newlabel()
B.false = S.next
S1.next = begin
S.code = label(begin) || B.code || label(B.true) || S1.code || gen('goto' begin)
Boolean expressions: value or control flow?
Boolean expressions

Boolean expressions can be evaluated
- to determine the flow of control
- for their value
6.6.6 Boolean values and jumping code

"S -> id = E; | if (E) S | while (E) S | S S

Nonterminal E governs the flow on control in $S \rightarrow \text{while} (E) \ S1$. The same nonterminal E denotes a value in $S \rightarrow id = E; [...]$"
6.6.6 Boolean values and jumping code

“Suppose that attribute E.n denotes the syntax-tree node for an expression E and that nodes are objects. Let method jump generate jumping code at an expression node, and let method rvalue generate code to compute the value of the node into a temporary.” [p. 408]
Value of Boolean expression

"When E appears in S -> while (E) S1, method jump is called at node E.n
[...]
When E appears in S -> id = E;, method rvalue is called at node E.n" [p. 408]
"If $E$ has the form $E_1 + E_2$, the method call $E.n.rvalue()$ generates code as discussed in section 6.4."

"$E \rightarrow E_1 + E_2$

$E.addr = \text{new Temp()}$
$E.code = E_1.code || E_2.code || \text{gen}(E.addr '=' E_1.addr '+' E_2.addr)" \[p. 379\]

"If $E$ has the form $E_1 \&\& E_2$ we first generate jumping code for $E$ and then assign true or false to a new temporary $t$ at the true and false exits, respectively, from the jumping code." \[p. 408\]

Translation of: $x = a < b \&\& c < d$

ifFalse a < b goto L1
ifFalse c < d goto L1
$t = \text{true}$
goto L2
L1:  $t = \text{false}$
L2:  $x = t$
Boolean expressions

Examples:  !X  X & Y  X | Y
Boolean expressions

Examples: !X  X & Y  X | Y

We will do short-circuit evaluation
Boolean expressions

Examples: \( \neg X \quad X \land Y \quad X \lor Y \)

We will do short-circuit evaluation

For example:

\[
\text{if } (X \lor (Y \land Z)) \text{ then } \{ A \} \text{ else } \{ B \}
\]

is translated as

\[
\text{if } X \text{ goto LA}
\]
\[
\text{ifFalse } Y \text{ goto LB}
\]
\[
\text{ifFalse } Z \text{ goto LB}
\]

LA: A

\[
\text{goto END}
\]

LB: B

END: (next instruction)
Boolean expressions

A concrete exercise - how is this translated?

if ( \( r < s \) | (r = s \& \& 0 < s) ) then \{ A \} else \{ B \}
Boolean expressions

- A concrete exercise - how is this translated?
- if ( r < s | (r = s & 0 < s)) then { A } else { B }

Here's a summary of the Intermediate Representation (IR) that we'll be using.

Three address code instructions (see 6.2.1, pages 364-5)

1. x = y op z
2. x = 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 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

© 2020 Carl Alphonse - Reproduction of this material is prohibited without the author's consent
Boolean expressions

A concrete exercise – how is this translated?

if \( r < s \) \( \lor \) \( (r = s \land 0 < s) \) then \{ A \} else \{ B \}

Exercise: try to come up with a suitable translation.

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. goto \( L \)
5. if \( x \) goto \( L \) / ifFalse \( x \) goto \( L \)
6. if \( x \) relop \( y \) goto \( L \)
7. function calls:
   - param \( x \)
   - call \( p, n \)
   - \( y = \) call \( p \)
   - return \( y \)
8. \( x = y[i] \) and \( x[i] = y \)
9. \( x = \Delta y, x = *y, *x = y \)

We'll start with these.
We'll spend significant time on function calls later.
We'll explore these as needed later on.
Boolean expressions

- A concrete exercise - how is this translated?

\[
\text{if } (r < s \mid (r = s \& 0 < s)) \text{ then } \{ A \} \text{ else } \{ B \}
\]

This has the same form as our example:

\[
\text{if } (X \mid (Y \& Z)) \text{ then } \{ A \} \text{ else } \{ B \}
\]

is translated as

- if \( X \) goto LA
- ifFalse \( Y \) goto LB
- ifFalse \( Z \) goto LB

LA:
- A
- goto END

LB:
- B

END: (next instruction)

Three address code instructions (see 6.2.1, pages 364-5)

1. \( x = y \text{ op } z \)
2. \( x = \text{ op } y \)  \hspace{1cm} (treat i2r and r2i as unary ops)
3. \( x = y \)
4. goto L
5. if \( x \) goto L / ifFalse \( x \) goto L
6. if \( x \) 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. \( x = \&y, x = *y, *x = y \)

© 2020 Carl Alphonse - Reproduction of this material is prohibited without the author's consent
Boolean expressions

- A concrete exercise - how is this translated?

- if \(( r < s \land (r = s \land 0 < s))\) then \{ A \} else \{ B \}

This has the same form as our example:

- if \((X \lor (Y \land Z))\) then \{ A \} else \{ B \}

is translated as

1. if \(X\) goto LA
2. ifFalse \(Y\) goto LB
3. ifFalse \(Z\) goto LB

LA:

- A
- goto END

LB:

- B

END: (next instruction)

Three address code instructions
(see 6.2.1, pages 364-5)

1. \(x = y \op z\)
2. \(x = \op y\) \hspace{1cm} (treat \text{i2r} and \text{r2i} as unary ops)
3. \(x = y\)
4. 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. \(x = \delta y, x = *y, *x = y\)

We'll start with these.

We'll spend significant time on function calls later.

We'll explore these as needed later on.

© 2020 Carl Alphonse - Reproduction of this material is prohibited without the author's consent
Boolean expressions

- A concrete exercise - how is this translated?

if ( r < s | (r = s & 0 < s)) then { A } else { B }

This has the same form as our example:

if ( r < s | (Y & Z)) then { A } else { B } is translated as

if r < s goto LA
ifFalse Y goto LB
ifFalse Z goto LB
LA: A
goto END
LB: B
END: (next instruction)

Three address code instructions (see 6.2.1, pages 364-5)

1. x = y op z
2. x = 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 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 = δy, x = *y, *x = y

We'll start with these.
We'll spend significant time on function calls later.
We'll explore these as needed later on.
Boolean expressions

A concrete exercise - how is this translated?

if ( r < s | (r = s & 0 < s)) then { A } else { B } is translated as

if r < s goto LA
ifFalse r = s goto LB
ifFalse Z goto LB
LA:   A
goTo END
LB:   B
END: (next instruction)

Three address code instructions (see 6.2.1, pages 364-5)

1. x = y op z   (treat i2r and r2i as unary ops)
2. x = op y
3. x = y
4. 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 = call p
   - return y
8. x = y[i] and x[i] = y
9. x = &y, x = *y, *x = y

This has the same form as our example:

if ( r < s | (r = s & Z)) then { A } else { B }

is translated as

if r < s goto LA
ifFalse r = s goto LB
ifFalse Z goto LB
LA:   A
goTo END
LB:   B
END: (next instruction)
Boolean expressions

- A concrete exercise - how is this translated?

if (r < s | (r = s & 0 < s)) then {A} else {B} is translated as

```
if r < s goto LA
ifFalse r = s goto LB
ifFalse 0 < s goto LB

LA:   A
      goto END
LB:   B
END: (next instruction)
```

Three address code instructions (see 6.2.1, pages 364-5)

1. x = y op z
2. x = 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 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

We'll start with these.

We'll spend significant time on function calls later.

We'll explore these as needed later on.
Backpatching

Allows jump targets to be filled in during a one-pass parse.

When (forward) jumps are needed, keep a list of where the addresses need to be inserted.

Once address is known, go back and fill in the address ("backpatching").
6.7 Backpatching

“For specificity, we generate instructions into an instruction array, and labels will be indices into this array.” [p. 410]

We have an instruction pointer (the first available index in the array), called nextinstr.
6.7 Backpatching

makelist(i) creates a new list containing only i, an index into the array of instructions; makelist returns a pointer to the newly created list.

merge(p1, p2) concatenates the lists pointed to by p1 and p2, and returns a pointer to the concatenated list.

backpatch(p, i) inserts i as the target label for each of the instructions on the list pointed to by p.