CSE443
Compilers

Dr. Carl Alphonce
alphonce@buffalo.edu
343 Davis Hall
Phases of a compiler

Intermediate Representation (IR): specification and generation

Figure 1.6, page 5 of text
Intermediate Representation (IR)

we'll use IR defined in textbook: the "three address code"
Three address code instructions
(see 6.2.1, pages 364-5)

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

What information does a type convey?

How is type information used during compilation?
type information

What information does a type convey?
- type indicates size
- type indicates storage location
  (a) primitives: either stack or heap
  (b) records: on heap (via pointer)
  (c) arrays: on heap (via pointer)
  (d) functions: code in static, locals on stack

How is type information used during compilation?
type information

- What information does a type convey?
  - type indicates size
  - type indicates storage location
    - (a) primitives: either stack or heap
    - (b) records: on heap (via pointer)
    - (c) arrays: on heap (via pointer)
    - (d) functions: code in static, locals on stack

- How is type information used during compilation?
  - determines how to lay out records, arrays, invocation records in memory
  - determines how to translate names in program to memory accesses
  - determines which instructions to use to manipulate values in memory

© 2020 Carl Alphonce - Reproduction of this material is prohibited without the author's consent
Sizes of types

- int: 32 bits (2's complement)
- real: 64 bits (IEEE 754)
- Boolean: 8 bits (TBD: machine dependent)
- character: 8 bit (ASCII)
- address: 64 bits
Sizes/layouts of values of types

- type string: 1 -> character
- 4 bytes + length of string * size of character (= 1 byte)
- # of dimensions is part of type

<table>
<thead>
<tr>
<th>size of dimension 1 (integer)</th>
<th>(0)</th>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
<th>(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 S V A X E S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

https://en.wikipedia.org/wiki/VAX
Array layout in memory

Two options:

- row-major
- column-major

Textbook discusses on page 382; row-major and column-major refer to two-dimensional arrays, but can be generalized for arrays with more dimensions.
Row-major array layout

What is the size of an $X$-dimensional array of type $T$?

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

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

(plus padding for real to get to proper boundary)

Example shows two-dimensional array (2 rows, 3 columns)
### Column-major array layout

What is the size of an $X$-dimensional array of type $T$?

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

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

Example shows two-dimensional array (2 rows, 3 columns)

<table>
<thead>
<tr>
<th>size of first dimension</th>
<th>size of second dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>first col</th>
<th>second col</th>
<th>third col</th>
</tr>
</thead>
<tbody>
<tr>
<td>a(0,0)</td>
<td>a(0,1)</td>
<td>a(0,2)</td>
</tr>
<tr>
<td>a(1,0)</td>
<td>a(1,1)</td>
<td>a(1,2)</td>
</tr>
</tbody>
</table>
Propagation of type information in parse tree
For the purposes of type checking the number of dimensions is relevant, but the size of each dimension is not.
Figure 6.16, p 375

For alpha, add space for size of each dimension.

What if type info comes after dimensions?

8 + array(2, arr(3, character))

\[ w = 8 + 6 \times 1 = 14 \]

\[ w = 2 \times 3 \]

\[ w = 6 \]

Character

\[ w = 1 \]
Planting a seed...

Q: if a and b are compatible array types, what are the semantics of a := b ?
Q: if a and b are compatible array types, what are the semantics of a := b ?

A: Copy the contents of b (an address) into a.
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 memory.
Variables and memory

- Every use of a variable is translated into an address by the compiler...
  
  ...but not an absolute address - we have no idea where in memory things will be loaded!

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

- Variables housed within each block have 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.
Arrays

What is the size of a multi-dimensional array of type T?

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

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

assume sizeOf(T) is 1

address \(a(0,0)\): offset 8
address \(a(0,1)\): offset 9
address \(a(0,2)\): offset 10
address \(a(1,0)\): offset 11

etc.
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(); }

AT RUNTIME

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

pop offset = 8 from stack

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

pop offset = 8 from stack
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(); }

offset = 0
offset = 4
offset = 8

{ ( integer : x , y )
push offset = 8 onto stack
offset = 8
offset = 16
offset = 24

{ ( real : x , z ) ... ... }
pop offset = 8 from stack
push offset = 8 onto stack
offset = 8
offset = 9
-offset = 9

{ ( boolean : y ; character : z )
offset = 10

... }
pop offset = 8 from stack

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 → ']' 
  { 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(); }

\[ \text{declaration-list} \]

AT RUNTIME

\[
\begin{array}{c}
\text{integer: } x \\
\text{integer: } y
\end{array}
\]

\[
\begin{array}{c}
\text{real: } x \\
\text{real: } z
\end{array}
\]

\[
\begin{array}{c}
\text{Boolean: } y \\
\text{character: } z
\end{array}
\]

© 2020 Carl Alphonce - Reproduction of this material is prohibited without the author's consent
**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:

\[
\begin{align*}
\text{dblock} & \rightarrow \left[ \begin{array}{l}
\{ \text{Env.push(st); st = new Env(); Stack.push(offset); offset = 0; } \\
\text{declaration-list } \} \\
\{ \text{dblock.type=record(st); dblock.width=offset; st=Env.pop(); offset=Stack.pop(); } \\
\end{array} \right]
\end{align*}
\]

**AT RUNTIME**

- \[ \text{offset = 0} \]
- \[ \text{push offset = 8 onto stack} \]
- \[ \begin{array}{c}
\text{offset = 8} \\
\text{offset = 16} \\
\text{offset = 24}
\end{array} \]
- \[ \text{pop offset = 8 from stack} \]
- \[ \text{push offset = 8 onto stack} \]
- \[ \begin{array}{c}
\text{offset = 8} \\
\text{offset = 9} \\
\text{offset = 10}
\end{array} \]
- \[ \text{pop offset = 8 from stack} \]
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(); }

AT RUNTIME

doctest

offset = 0
offset = 4
offset = 8

{ ( integer : x , y )
  push offset = 8 onto stack
  offset = 8
  offset = 16
  offset = 24

  { ( real : x , z ) ... ... }

pop offset = 8 from stack
push offset = 8 onto stack
offset = 8
offset = 9

{ ( Boolean : y ; character : z )
  :offset = 8
  offset = 9
  offset = 10
  offset = 11

pop offset = 8 from stack

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

\[
\begin{align*}
\{ & \text{[ Boolean : } a \text{ ; integer : } x \text{ ; character : } c \text{ ; real : } y \} \\
\{ & \text{[ character : } d \text{ ; integer : } r, s \} \ldots \} \\
\{ & \text{[ Boolean : } f, g \text{ ; real : } t \text{ ; character : } h \} \ldots \} \\
\}
\end{align*}
\]

"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 \} \ldots \}
  \{ Boolean : f , g ; real : t ; character h \} \ldots \}
\}

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

Blocks are not aligned.

A block of size N bytes typically needs be aligned to an address divisible by N, where N is an integral power of 2 \((1, 2, 4, 8)\)
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.
IR: a motivating example
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 -> while (E) S1. The same nonterminal E denotes a value in S -> id = E; [...]"

[p. 408]
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]
"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 E1 + E2, the method call E.n.rvalue() generates code as discussed in section 6.4." [p. 408]

\[E \rightarrow E1 + E2\]
\[E.\text{addr} = \text{new Temp()}\]
\[E.\text{code} = E1.\text{code} \| E2.\text{code} \| \text{gen}(E.\text{addr} '=' E1.\text{addr} '+' E2.\text{addr})]\] [p. 379]

"If E has the form E1 && E2 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 = true
goto L2
L1: t = false
L2: x = t
Boolean expressions

Examples: \( \neg x \quad x \land y \quad x \lor y \)
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

For example:

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

is translated as

\[
\begin{align*}
\text{if } X & \text{ goto LA} \\
\text{ifFalse } Y & \text{ goto LB} \\
\text{ifFalse } Z & \text{ goto LB}
\end{align*}
\]

LA: A

\[
goto \text{ END}
\]

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 \( r < s \lor (r = s \land 0 < s) \) 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 2
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. \( x = \) dy, \( x = *y, *x = y \)

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