Phases of a compiler

Intermediate Representation (IR): specification and generation

Figure 1.6, page 5 of text
CSE443 Compilers

Lecture 23

Part 1
Variables

- **type** - size, encoding, location (e.g. stack vs heap)
- **name** - refers to variable in high level program
- **value** - the contents of the variable, at runtime
- **location** - where in memory the variable exists
- **scope** - the part of the program in which the variable may be used
- **lifetime** - the period of time during execution of the program that the variable exists in memory
Variables

- type - in symbol table
- name - in symbol table
- value - at runtime (except initialization)
- location - offset determined during compilation
- scope - determined by program structure, reflected in symbol table structure
- lifetime - determined at runtime
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 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.
### Arrays

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

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

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

**assume sizeof(T) is 1**

<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>0</td>
<td>3</td>
</tr>
</tbody>
</table>

| address for $a(0,0)$: offset 8 | $a(0,0)$ |
| address for $a(0,1)$: offset 9 | $a(0,1)$ |
| address for $a(0,2)$: offset 10 | $a(0,2)$ |
| address for $a(1,0)$: offset 11 | $a(1,0)$ |

<table>
<thead>
<tr>
<th>first row</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a(1,0)$</td>
</tr>
<tr>
<td>$a(1,1)$</td>
</tr>
<tr>
<td>$a(1,2)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>second row</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a(1,0)$</td>
</tr>
<tr>
<td>$a(1,1)$</td>
</tr>
<tr>
<td>$a(1,2)$</td>
</tr>
</tbody>
</table>

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

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: keep a running offset, but remember old offset when entering embedded scope.

```plaintext
dblock → '['
  { Stack.push(offset); }  
  { declaration-list ']' }  
  { offset=Stack.pop(); }  
```

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

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

offset = 4
push offset = 16 onto stack
offset = 16
push offset = 24 onto stack
offset = 24

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

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

offset = 10
pop offset = 8 from stack
offset = 10
push offset = 8 onto stack
offset = 10
```
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, we can do better:

\[
\text{dblock} \rightarrow \left[ \begin{array}{l}
\{ \text{Env} \text{.push}(\text{st}); \text{st} = \text{new Env}(); \text{Stack} \text{.push}(\text{offset}); \text{offset} = 0; \} \\
\text{declaration-list} \}\right]
\]

\[
\{ \text{dblock} \text{.type=} \text{record}(\text{st}); \text{dblock} \text{.width=} \text{offset}; \text{st=} \text{Env} \text{.pop}(); \text{offset=} \text{Stack} \text{.pop}(); \}
\]
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:

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
{ (integer : x , y ) }
op offset = 8 from stack
push offset = 8 onto stack
{ (real : x , z ) ... ... }
op offset = 8 from stack
push offset = 8 onto stack
{ (Boolean : y ; character : z ) ... ... }
op offset = 8 from stack

© 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:

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

**AT RUNTIME**

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

\[
\text{dblock} \rightarrow '][' \{ \begin{array}{l}
(\text{integer } : x, y) \{ (\text{real } : x, z) \ldots \ldots \}
\text{push offset = 8 onto stack}
\text{pop offset = 8 from stack}
\text{push offset = 8 onto stack}
(\text{boolean } : y, \text{character } : z) \ldots \ldots
\text{pop offset = 8 from stack}
\end{array}\}
\]
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
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; }
declarations-list ']

{ dblock.type=record(st); dblock.width=offset; st=Env.pop(); offset=Stack.pop(); }

\[\begin{align*}
\text{offset} &= 0 \\
\text{offset} &= 4 \\
\text{offset} &= 8 \\
\text{offset} &= 8 \\
\text{offset} &= 12 \\
\text{offset} &= 16 \\
\text{offset} &= 24 \\
\text{offset} &= 8 \\
\text{offset} &= 8 \\
\text{offset} &= 9 \\
\text{offset} &= 10 \\
\text{offset} &= 8 \\
\text{offset} &= 8
\end{align*}\]

AT RUNTIME
CSE4433 Compilers

Lecture 23

Part 2
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

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

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

\[
\begin{cases}
    \text{Boolean: } a \\
    \text{integer: } x \\
    \text{character: } c \\
    \text{real: } y
\end{cases}
\]

\[
\begin{cases}
    \text{character: } d \\
    \text{integer: } r, s
\end{cases}
\]

\[
\begin{cases}
    \text{Boolean: } f, g \\
    \text{real: } t \\
    \text{character: } h
\end{cases}
\]

"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

\{ [ \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 \} \}

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 ] ... }

}