Phases of a compiler

Intermediate Representation (IR): specification and generation

Figure 1.6, page 5 of text
Review

- Project 1: char stream → LEXER → token stream
- Project 2: PARSER builds symbol table, checks for undefined or multiply defined names from token stream.
- Project 3: PARSER will also perform type checking and generate intermediate code.
Our language
(use name equivalence)

- **pre-defined types:**
  - **primitive types:** integer, real, Boolean, character
  - **composite type:** string
- **user-defined types:**
  - **record types have names**
    - `type rec : [ real : x , y ]`
  - **array types have names**
    - `type arr : 2 -> string`
  - **function types have names**
    - `type fun : ( real : x ) -> rec`
Recursive records

Recursive functions

A record type must allow a component to be of the same type as the type itself:

type Node: [ integer datum:=0 ; Node rest:=null ]
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. \( x = \text{ op } y \)  \( \text{ (treat i2r and r2i as unary ops)} \)
3. \( x = y \)
4. \( \text{goto L} \)
5. \( \text{if } x \text{ goto L / ifFalse } x \text{ goto L} \)
6. \( \text{if } x \text{ relop } y \text{ goto L} \)
7. function calls:
   - param x
   - call p, n
   - y = call p
   - return y
8. \( x = y[i] \text{ and } x[i] = y \)
9. \( x = \&y, x = *y, *x = y \)
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. \( x = &y, x = *y, *x = y \)

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?
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?
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
Sizes of types

- int: 32 bits (2's complement)
- real: 64 bits (IEEE 754)
- Boolean: 8 bits (TBD: machine dependent)
- character: 8 bit (ASCII)
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</th>
<th>(0)</th>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
<th>(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(integer)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>V</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>E</td>
<td></td>
<td>S</td>
<td></td>
</tr>
</tbody>
</table>
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*4 bytes

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

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

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

<table>
<thead>
<tr>
<th></th>
<th>first row</th>
<th>second row</th>
</tr>
</thead>
<tbody>
<tr>
<td>a(0,0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a(0,1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a(0,2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a(1,0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a(1,1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a(1,2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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: $(\prod_{i \in X} S_i) \times \text{sizeOf}(T)$

Example shows two-dimensional array (2 rows, 3 columns)
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.
For alpha, add space for size of each dimension.

What if type info comes after dimensions?

$8 + \text{array}(2, \text{arr}(3, \text{character}))$

$w = 8 + 6 \times 1 = 14$

$w = 2 \times 3$

$w = 6$

character

$w = 1$

$[2]$

$[3]$

© 2020 Carl Alphonce - Reproduction of this material is prohibited without the author's consent
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: We won't answer this now, but think about it...
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.
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 \(\text{sizeof}(T)\) 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>2</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

base address and offset (0) for size of first dimension

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

etc.

first row

second row
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.

Scopes

\[ \text{dblock} \rightarrow \['\]
\{ 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 running offset, but remember old offset when entering embedded scope.

offset = 0
\{ [ integer : x , y ] \}
push offset = 8 onto stack
offset = 8
\{ [ real : x , z ] ... ... \}
pop offset = 8 from stack
push offset = 8 onto stack
offset = 8
\{ [ Boolean : y ; character : z ] ... ... \}
op offset = 8 from stack
\}

integer: x
integer: y
integer: x
integer: y
real: x
real: x
real: z
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(); }

\[
\begin{align*}
\text{offset} &= 0 \\
\text{integer: x} & \quad \text{offset} = 4 \\
\text{integer: y} & \quad \text{offset} = 8 \\
\text{push offset} = 8 \text{ onto stack} \\
\text{offset} &= 16 \\
\text{real: x} & \quad \text{offset} = 24 \\
\text{offset} &= 28 \\
\text{real: z} & \quad \text{offset} = 32 \\
\text{push offset} = 8 \text{ onto stack} \\
\text{offset} &= 36 \\
\text{integer: x} & \quad \text{offset} = 40 \\
\text{integer: y} & \quad \text{offset} = 44 \\
\text{offset} &= 48 \\
\text{Boolean: y} & \quad \text{offset} = 56 \\
\text{character: z} & \quad \text{offset} = 60 \\
\text{offset} &= 64 \\
\text{push offset} = 8 \text{ onto stack} \\
\text{offset} &= 72 \\
\text{offset} &= 76 \\
\end{align*}
\]

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(); }
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 
{  integer : x , y } 
push offset = 8 onto stack

 offset = 4 
{  real : x , z } 
offset = 8

 offset = 16 
{  boolean : y ; character : z } 
offset = 24

 AT RUNTIME

offset = 8

push offset = 8 onto stack

offset = 8

pop offset = 8 from stack

offset = 8

push offset = 8 onto stack

offset = 8

offset = 9

offset = 10

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(); }

integer: x
integer: y
Boolean: y
character: z

AT RUNTIME

offset = 8
push offset = 8 onto stack
offset = 16
offset = 24
pop offset = 8 from stack
push offset = 8 onto stack
offset = 9
offset = 10
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

push offset = 8 onto stack

offset = 8
offset = 16
offset = 24

pop offset = 8 from stack

push offset = 8 onto stack

offset = 8
offset = 9
offset = 10

pop offset = 8 from stack

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

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

"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

{ [ 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 : $h$ ] ... }

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.
Offsets and alignment in the project

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