Assignment 10

Contents

1. Background

In this assignment, we add various Scheme datatypes and primitives to our language, which is a major step in making the language more Scheme-like.

2. Scheme Subset 10

Here is a grammar for the subset of Scheme we'll be handling this week:

Program	⟶	`(letrec ([`*label* `(lambda (`*uvar*`*)` *Value*`)]*)` *Value*`)`
Value	⟶	*label*
	|	*uvar*
	|	`(quote` *Immediate*`)`
	|	`(if` *Pred* *Value* *Value*`)`
	|	`(begin` *Effect*`*` *Value*`)`
	|	`(let ([`*uvar* *Value*`]*)` *Value*`)`
	|	`(`*value-prim* *Value*`*)`
	|	`(`*Value* *Value*`*)`
Pred	⟶	`(true)`
	|	`(false)`
	|	`(if` *Pred* *Pred* *Pred*`)`
	|	`(begin` *Effect*`*` *Pred*`)`
	|	`(let ([`*uvar* *Value*`]*)` *Pred*`)`
	|	`(`*pred-prim* *Value*`*)`
Effect	⟶	`(nop)`
	|	`(if` *Pred* *Effect* *Effect*`)`
	|	`(begin` *Effect*`*` *Effect*`)`
	|	`(let ([`*uvar* *Value*`]*)` *Effect*`)`
	|	`(`*effect-prim* *Value*`*)`
	|	`(`*Value* *Value*`*)`
Immediate	⟶	*fixnum* | () | `#t` | `#f`

Unique variables (*uvar*), labels (*label*), and restrictions upon unique variables and labels are as in the Assignment 9 subset.

A *fixnum* is an exact integer in a machine-dependent range, which can be determined from the helpers.ss `fixnum-bits`.

Valid *value-prim*, *pred-prim*, and *effect-prim* names and argument counts are given by the following table.

category	name	arguments
value	`+`	2
	`-`	2
	`*`	2
	`car`	1
	`cdr`	1
	`cons`	2
	`make-vector`	1
	`vector-length`	1
	`vector-ref`	2
	`void`	0
pred	`<=`	2
	`<`	2
	`=`	2
	`>=`	2
	`>`	2
	`boolean?`	1
	`eq?`	2
	`fixnum?`	1
	`null?`	1
	`pair?`	1
	`vector?`	1
effect	`set-car!`	2
	`set-cdr!`	2
	`vector-set!`	3

3. Object Representation

The heart of our object representation is the *ptr* (which rhymes with "footer.") A ptr is, simply, an encoding of a Scheme object as an integer. For immediate objects, such as fixnums, the ptr is a tagged version of the object. For heap-allocated objects, such as pairs, the ptr is a tagged pointer to the object.

Why are objects tagged? It would be more natural to represent, e.g., fixnums as their integer equivalents and pairs as true pointers. This is not generally possible, however, since we need to be able to distinguish, at run time, pairs from fixnums and both pairs and fixnums from all other types of objects. Otherwise, we could not implement type predicates like `pair?` and `fixnum?`. We would also have problems down the road with type checking and garbage collection. So we give each kind of object a different tag. Fortunately, the tagging mechanism we use adds little overhead.

We will use a 64-bit *low-tag* representation for our ptrs, in which the tag is stored in the low-order three bits of a 64-bit word. So that we can use three bits for the tag, we align each true pointer on a 64-bit (8-byte, double-word) boundary, leaving three bits that are always zero at the bottom of each true pointer. These three bits can be set to any value and still identify the true address as long as we zero them out or otherwise adjust for their presence. Another way to look at this is that each true address is a multiple of eight, so we can use all of the numbers in between, e.g., addr + 1 thorugh addr + 7 to identify different types of objects.

We call the low three bits of a ptr the *primary tag*. One primary tag is dedicated to fixnums, one to "other immediates," one to pairs, one to procedures, and one to vectors. The remaining three primary tags out of the eight possible tag values are unassigned.

Fixnums are assigned tag 000_2. The ptr representation of a fixnum is merely a 61-bit integer value, shifted left by three bits, or multiplied by 8. For example, the fixnum 7 (111_2) is represented by the ptr 56 (111000_2). It turns out to be convenient for the low-order bits to be zero and for the amount of the multiplier to be the same as the number of bytes in a word, for reasons that we will get to later.

Pairs, procedures, and vectors are our only heap-allocated objects. A ptr is obtained from the true address of a heap-allocated object by adding the tag to the true address. For example, the ptr representation of an object with tag `5` at true address `2000` is `2005`. Naturally, this means that the true address can be obtained from the ptr by subtracting the tag.

All immediates other than fixnums, i.e., booleans, the empty list, and void, share a single primary tag. In order for them to share a single primary tag, we need to use additional bits to distinguish them, i.e., a *secondary tag*. We use the remaining five bits of the least-significant byte as the secondary tag. Another way to look at it is that, for non-fixnum immediates, we use the entire least-significant byte as the tag. In fact, the least-significant byte defines the entire value of the constant for each of these immediates.

We can determine if we have a particular type of object by applying the appropriate mask (using `logand`) and comparing the resulting value with the tag for that type of object. For example, to determine if we have a pair, we apply a mask of 111_2 and compare the result with the tag for pair.

We have three unused primary-tag values. If we had more than three additional heap-allocated types, we would not have enough primary tags to support them. In this case, we would set aside one of the tags as an escape tag. An object with this primary type would contain the actual tag (the secondary tag) in the first word of its heap-allocated component. This mechanism yields a virtually unlimited number of tag values and can thus be used to support indefinite numbers of user-defined types as well.

To help make this more concrete, here is one possible assignment of primary tags, shown in base two.

- `000`: fixnums
- `001`: pairs
- `010`: procedures
- `011`: *unused*
- `100`: *unused*
- `101`: *unused*
- `110`: non-fixnum immediates
- `111`: vectors

With this assignment of primary tags, a procedure at true address 80000 is represented by the ptr 80002, which in binary looks like

```
10011100010000010
```

Here is a possible assignment of secondary tags for non-fixnum immediates, with the entire low-order byte shown.

- `00000110`: `#f`
- `00001110`: `#t`
- `00010110`: `()`
- `00011110`: void

Many other assignments are possible. Indeed, the commitments to three low-order tag bits and a word size of 64 bits (eight bytes) are not set in stone. Because these things may change over time, it is good practice not to build any tag-dependent constants into the compiler. Instead, use the symbolic names defined in helpers.ss. For example, the tag value for pair is given by the value of the variable `tag-pair`, and the amount by which fixnums are shifted is given by `shift-fixnum`. Take a look at the definitions of these variables and the related variables in helpers.ss to familiarize yourself with what's there.

It is a good idea to test your compiler with alternative values for these variables to make sure that you have not built in particular tag values.

4. Things to do

To handle the replacement of UIL integers and UIL primitives with Scheme datatypes and Scheme primitives, along with the corresponding grammar changes, we must adjust the verifier and add a new pass, `specify-representation`, to translate the Scheme datatypes and primitives into UIL datatypes in primitives.

4.1. `verify-scheme`

This pass must be updated to reflect the changes in the language.

4.2. `specify-representation`

The goal of this pass is to convert all Scheme datatypes to their ptr equivalents so they are represented as integers in the output language, and at the same time, translate calls to Scheme primitives into calls to UIL primitives.

Although this pass handles all Scheme datatypes, it is useful to consider its responsibilities in three logical parts: (1) the handling of immediate data and operators on immediate data; (2) the handling of non-immediate (heap-allocated) data and operators on non-immediate data; and (3) the handling of type and equivalence predicates.

Handling immediate datatypes:

The immediate datatypes in our language are fixnums, booleans, the empty list, and the void object. The operators that deal specifically with fixnums are `*`, `+`, `-`, and the relational operators, e.g., `<` and `=`. Our language does not have any that deal specifically with the other immediate datatypes.

Converting boolean constants, the empty list, and void into their ptr equivalents is trivial. `#f` is converted into the value of the helpers.ss variable `$false`, `#t` to `$true`, `()` to `$nil`, and void to `$void`. (Although void does not appear as a quoted constant, the `void` value primitive should be treated as a void constant.)

Converting fixnums into their ptr equivalents is almost as easy: simply shift the fixnum value left by `shift-fixnum`.

Because of our choice of representation for fixnums, most of our primitive operations on fixnums need no adjustment. Assuming `shift-fixnum` is 3, each fixnum x is effectively represented by the integer $8x$. For `+` and `-`, no adjustment is necessary, since $8x + 8y = 8(x + y)$ and $8x - 8y = 8(x - y)$. Similarly, for `<`, `=`, etc., no adjustment is necessary, since $8x < 8y$ if and only if $x < y$, and $8x = 8y$ if and only if $x = y$, etc.

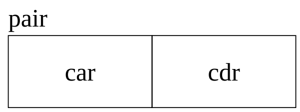
Only for `*` do we need to make an adjustment. We do need one for `*`, since $8x \cdot 8y = 64(x \cdot y)$. So we first shift one of the operands back to the right, noting that $8x \cdot y = 8(x \cdot y)$. If either argument is a constant, we can perform this shift at compile time; in this common case, there is no run-time overhead for the shift. Thus, for `(*` e_1 e_2`)` where neither e_1 nor e_2 is constant, we produce `(*` e_1 `(sra` e_2 `3))`, and for `(*` e `(quote` n`))` or `(* (quote` n`)` e`)` we produce `(*` e m`)`, where m is n shifted right by 3. In the run-time call to `sra`, the constant `3` is the actual value of `shift-fixnum`, not its ptr equivalent.

While most of our tag assignments are arbitrary, the decision to use a zero tag for fixnums was made precisely so that arithmetic operations are largely unaffected by the tagging. The value of `shift-fixnum` doesn't matter: a different shift would result in a different multiplier, but all of the same properties would still hold. Remember to use `shift-fixnum`, rather than constants like `3` and `8`, in your code.

Handling non-immediate (heap-allocated) datatypes:

There are no non-immediate constants in our language (yet), so we do not need to worry about their conversion to UIL constants. Also, we have not yet added first-class procedures, so the only heap-allocated datatypes we have at this point are pairs and vectors. Thus, we need focus only on handling the operators that create, access, and modify pairs and vectors, i.e., `cons`, `make-vector`, `car`, `cdr`, `vector-length`, `vector-ref`, `set-car!`, `set-cdr!`, and `vector-set!`. We need to convert each of these into code that employs the UIL primitives for allocating and manipulating memory, i.e., `alloc`, `mref`, and `mset!`.

Let's consider pairs first. A pair is represented by a ptr whose true address equivalent points to a two-word block of memory, with the car field possibly (but not necessarily) preceding the cdr field.



The accessors `car` and `cdr` are translated into calls to `mref` with different byte offsets. The byte offset of the car field of a pair from the true address of the pair is given by the helpers.ss variable `disp-car`. Simply adding `disp-car` to the ptr representation of a pair won't do, however, since the ptr is itself offset from the true address by `tag-pair`. Thus, the byte offset of the car field is determined by subtracting `tag-pair` from `disp-car`, i.e.,

```
(car e)  →  (mref e offset-car)
```

where *offset-car* is the value of `(- disp-car tag-pair)`. `cdr` is translated similarly, using `disp-cdr` instead of `disp-car`. The mutators `set-car!` and `set-cdr!` are converted into calls to `mset!`, using the same byte offsets as for `car` and `cdr`. For example:

```
(set-car! e1 e2)  →  (mset! e1 offset-car e2)
```

To allocate a pair, we simply invoke `alloc` as follows:

```
(alloc size-pair)
```

where *size-pair* is the value of the helpers.ss variable `size-pair`. `alloc` returns the address of the allocated block. We convert this address into a ptr by adding the pair tag to the address:

```
(+ (alloc size-pair) tag-pair)
```

We now have a pair, but we haven't yet filled it up. We do this with the `mset!` equivalents of `set-car!` and `set-cdr!`.

```
(cons e1 e2)  →
  (let ([tmp-car e1] [tmp-cdr e2])
    (let ([tmp (+ (alloc size-pair) tag-pair)])
      (begin
        (mset! tmp offset-car tmp-car)
        (mset! tmp offset-cdr tmp-cdr)
        tmp)))
```

We could perform the following simpler transformation.

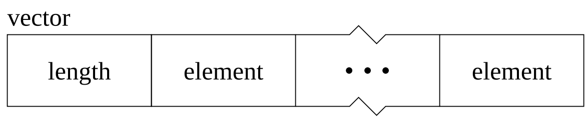
```
(cons e1 e2)  →
  (let ([tmp (+ (alloc size-pair) tag-pair)])
    (begin
      (mset! tmp offset-car e1)
      (mset! tmp offset-cdr e2)
      tmp))
```

This would violate Scheme's applicative-order semantics, however, since the pair would be allocated before the `cons` arguments. This is not detectable in our current subset of Scheme, but it will be if we decide to add `call/cc`.

To make this a bit more concrete, let's use the actual values of `size-pair`, `tag-pair`, `disp-car`, and `disp-cdr` from helpers.ss.

```
(cons e1 e2)  →
  (let ([tmp-car e1] [tmp-cdr e2])
    (let ([tmp (+ (alloc 8) 1)])
      (begin
        (mset! tmp -1 tmp-car)
        (mset! tmp 3 tmp-cdr)
        tmp)))
```

Handling vectors is slightly more complicated. A vector contains a length field followed by a sequence additional fields containing the elements of the vector..



The `vector-length` primitive may be handled in the same manner as `car`, using `tag-vector` and `disp-vector-length` in place of `tag-pair` and `disp-car`.

The UIL code for `vector-ref` is a bit more involved than the UIL code for `car`, since we now have an index to deal with. This index must be added to the offset of the first element to determine the offset of the selected element. The general case, where the index is not known at compile time, is given below.

```
(vector-ref e1 e2)  →  (mref e1 (+ offset-vector-data e2))
```

where *offset-vector-data* is the value of `(- disp-vector-data tag-vector)`. When the index is known at compile time, we can avoid the run-time addition by performing it at compile time.

```
(vector-ref e1 k)  →  (mref e1 n)
```

where n is the value of `(+ (- disp-vector-data tag-vector)` k`)`.

Is this really correct? We know that `mref` requires a byte offset, yet our vector indices are given as element numbers, which are essentially word offsets. Or are they? Here is where the representation of fixnums helps us once again. We have arranged for the fixnum multiplier to be precisely the number of bytes in a word, so that when we add the fixnum index, we're actually adding w times the actual index, where w is the number of bytes in a word. In this case, the tagging actually saves us an operation.

Handling `vector-set!` is similar to handling `vector-ref`, though an `mset!` call is produced rather than an `mref` call.

To allocate a vector, we must compute the size in much the same manner as we compute `mref` offsets for `vector-ref`. Otherwise, the allocation of vectors is similar to the allocation of pairs. Again, we have two cases: one in which the size is a constant and the other in which it is not. The constant-size case is shown below.

```
(make-vector k)  →
  (let ([tmp (+ (alloc n) ,tag-vector)])
    (begin
      (mset! tmp offset-vector-length k)
      tmp))
```

where *offset-vector-length* is the value of `(- disp-vector-length tag-vector)`, and n is the value of `(+ disp-vector-data` k`)`.

The variable-size case is similar.

```
(make-vector e)  →
  (let ([tmp1 e])
    (let ([tmp2 (+ (alloc (+ disp-vector-data tmp1)) tag-vector)])
      (begin
        (mset! tmp2 offset-vector-length tmp1)
        tmp2)))
```

Handling type and equivalence predicates:

Most of the remaining primitives are type predicates and are implemented, as described in Section 3, by isolating the appropriate set of tag bits and comparing them against the appropriate tag. The UIL primitive `logand` is used to isolate the tag, and `=` is used to perform the comparison:

```
(= (logand e mask) tag)
```

The mask and tag values for each *type* are given in the machine definition as `mask-`*type* and `tag-`*type*.

The `null?` predicate could be handled in the same manner as the other type predicates, but it is easier to treat it as a call to `eq?` with the second argument an implicit quoted empty list, i.e., treat `(null? e)` as `(eq? e '())`.

This leaves only `eq?`, which is easy. With all Scheme objects represented as integers (specifically, ptrs), `eq?` can simply be converted into `=`.

```
(eq? e1 e2)  →  (= e1 e2)
```

Summary:

The following items summarize what must be done on this pass:

- Convert quoted fixnums and other immediates into their unquoted ptr equivalents, using the values of the appropriate helpers.ss variables.
- Adjust one multiplication operand, at compile time if possible, otherwise at run time, using the helpers.ss variable `shift-fixnum`.
- Convert calls to `cons` and `make-vector` into calls to `alloc`. Take advantage of constant lengths in `make-vector`, where possible.
- Convert calls to `car`, `cdr`, `vector-length`, and `vector-ref` into calls to `mref`. Take advantage of constant indices in `vector-ref`, where possible.
- Convert calls to `set-car!`, `set-cdr!`, and `vector-set!` into calls to `mset!`. Take advantage of constant indices in `vector-set!`, where possible.
- Expand the type predicates `fixnum?`, `fixnum?`, `pair?`, and `vector?` into calls to `logand` and `=`, inserting the appropriate values of the symbolic mask and tag constants.
- Convert calls to `eq?` into calls to `=`. Treat calls to `null?` as calls to `eq?` with the empty list as one argument, hence convert them as well into calls to `=`.

That's it! A lot of code is involved to handle all of the cases, but with some good abstractions, many of the cases can be treated similarly.

5. Boilerplate and Run-time Code

The boilerplate code does not change. The run-time code in the updated runtime.c now assumes that the value returned from Scheme is a Scheme object, and it prints the object using a rudimentary Scheme printer, so that we can see actual lists, vectors, booleans, etc., in the output.

6. Testing

A small set of invalid and valid tests for this assignment have been posted in tests10.ss. You should make sure that your compiler passes work at least on this set of tests.

7. Coding Hints

Before starting, study the output of the online compiler for several examples.

Use `make-begin` to avoid nested `begin` expressions.