CSC 530 Lecture Notes Week 6

Discussion of Assignment 3, Questions 1 and 2

Introduction to Denotational Semantics

I. **Turingol Highlights**

A. Semantics define compilation of a TM language into quintuples.

B. Turingol semantics are *compiled*, SIL semantics are *interpreted*.

C. The form of instruction in the Turingol TM is:

Turingol, cont'd

$$< p , A , c , d , q >$$

where

p = present state

A = symbol scanned

c = symbol written

d = tape movement direction

q = next state

Turingol, cont’d

D. Attributes **Symbol** and **label**

1. Used as *symbol tables*, similar to env and store.

2. Store bindings of ident with value.

3. Here program names with TM-level values.

Turingol, cont'd

4. E.g., "**tape alpha is** point, blank, one, zero"

text(id)	**symbol(text(id))**
"point"	**.**
"blank"	**B**
"zero"	**1**
"one"	**0**

Turingol, cont'd

5. Similarly for statement labels.

text(id)	**label(text(id))**
test	q_2
carry	q_4
realign	q_7

Turingol, cont'd

E. Example 4.1 on page 137.

Source string	**TM quintuple**
print point	$<q_0$, s, ., 0, $q_1>$ where s = {B,0,1,.}
goto carry	$<q_1$, s, s, 0, $q_4>$

. . ._

Turingol, cont'd

F. Additional notes

1. Σ must be fully processed before any instructions.

2. **newsymbol** is Lisp's *gensym*.

3. **define** and **include** maintain set property.

II. Specifics for Assignment 3

A. For question 1, answer in terms of semantic attributes *not* TM states.

B. For question 2:

1. Make explicit attr dependencies.
2. **Label** most interesting.
3. Focus on the semantic definition technique, not TMs.

Now on to Denotational Semantics

III. **Reading: Papers 17-22, emphasis on 20**

IV. **Introductory comparison of Knuth-style semantics with Tennent-style**

A. In Knuth, rule eval strategy not explicitly specified.

B. In denotational, eval with formal function evaluation.

   1. Amounts to depth-first traversal

   2. Function args expressed in terms of syntactic constituents.

   3. Analog of passing attributes is passing function args.

Intro comparison, cont'd

4. Multiple eval passes based on one full-pass function invoking another.

5. Eval functions are first-call objects.

   a. We don/t need `functionize`

   b. No attributed parse trees.

6. Also, looping is more mathematical, using *fixpoints*.

C. More examples to follow.

V. Data domains, Tennent Ch 3

A. *Data domains* are the denotational analog of attribute type definitions.

B. As with attribute grammars, domain constructions are used for:

1. Defining definitional datatypes.

2. Model higher-level data.

Data domains, cont'd

C. Summary of what domain constructions model:

1. Product domains are *records*

2. Sum domains are *unions* (aka, *variant records*).

Data domains, cont'd

3. Function domains model *arrays* and other forms of *tables*.

4. Also to model the *value* of a procedure body (i.e., a lambda expr).

5. As in Lisp, recursive domains provide same capabilities as *pointers*.

VI. Binary numeral example

A. Tennent Ch 13 starts with it.

1. Knuth paper has similar example.

2. We’ll compare three semantic approaches --

   denotational, attribute grammars, and operational.

Binary numbers, cont'd

B. Denotational definition

Abstract syntax: N $\in$ **Nml** = binary numerals
I $\in$ **Int** = binary integers
F $\in$ **Frac** = binary fractions

N ::= I . F
I ::= B | I B
F ::= B | B F
B ::= 0 | 1

Semantic domain: Z = real numbers

Binary numbers, cont'd

Semantic functions: $\mathcal{N}$: **Nml** $\rightarrow$ **Z**

$\mathcal{I}$: **Int** $\rightarrow$ **Z**

$\mathcal{F}$: **Frac** $\rightarrow$ **Z**

$$\mathcal{N}[\![\mathrm{I}\ .\ \mathrm{F}]\!] = \mathcal{I}[\![\mathrm{I}]\!] + \mathcal{F}[\![\mathrm{F}]\!]$$

$$\mathcal{I}[\![\mathrm{I\ B}]\!] = 2 * \mathcal{I}[\mathrm{I}] + \mathcal{I}[\![\mathrm{B}]\!]$$

$$\mathcal{I}[\![0]\!] = 0$$

$$\mathcal{I}[\![1]\!] = 1$$

$$\mathcal{F}[\![\mathrm{B\ F}]\!] = \mathcal{F}[\![\mathrm{B}]\!] + \mathcal{F}[\![\mathrm{f}]\!]\ /\ 2$$

$$\mathcal{F}[\![0]\!] = 0$$

$$\mathcal{F}[\![1]\!] = 1/2$$

Binary numbers, cont'd

C. Attribute grammar definition

Atrribute	Description
v	Real number decimal value of the binary number.

Grammar and semantic equations:

```
N ::= I . F  {$$.v = $1.v + $3.v};
I ::= I B    {$$.v = 2 * $1.v + $2.v};
I ::= B      {$$.v = $1.v};
F ::= B F    {$$.v = $1.v + $2.v / 2};
F ::= B      {$$.v = $1.v / 2};
B ::= 1      {$$.v = 1};
B ::= 0      {$$.v = 0};
```

Binary numbers, cont'd

D. Operational definition

```
; Operational semantics for binary numbers, patterned after the attribute
; grammar and denotational definitions in
; ../semantics-expamples/binary-numbers{attr,deno}, q.q.v.
;
; Syntactically, a binary number is represented as a list of 0's and 1's, with
; an optional decimal point.  E.g., ( 1 1 0 1 . 0 1 ).

(defun main ()
    (let ((number (read)))
        (eval-binary-number number)
    )
)

(defun eval-binary-number (number)
    (let* ((integer-value (eval-integer-part number 0))
           (number (move-upto-dot number))
           (fractional-value (eval-fractional-part number 0)))
        (+ integer-value fractional-value)
    )
)

(defun eval-integer-part (number val)
    (cond ( (or (null number) (eq (car number) '.))
            val )
          ( t
            (let* ((val (+ (* 2 val) (car number))))
                (eval-integer-part (cdr number) val)) )
    )
)

(defun eval-fractional-part (number val)
    (cond ( (null number)
            val )
          ( t
            (let* ((val (/ (eval-fractional-part (cdr number) val) 2.0)))
                (+ (/ (car number) 2.0) val)) )
    )
)

(defun move-upto-dot (number)
    (cond ( (null number)
            nil )
          ( (eq (car number) '.)
            (cdr number) )
          ( (or (eq (car number) 0) (eq (car number) 1))
            (move-upto-dot (cdr number)) )
    )
)
```

Binary numbers, cont'd

E. Some observations

1. Syntax in attr def slightly more verbose

2. Heart of attribute grammar and denotational semantics is the same.

3. Operational semantics is considerably bulkier.

VII. **Notational conventions**

A. Double square brackets enclose syntactic operands (all of parsing).

B. ? is the "union tag test" operator.

1. E.g., b?**T**, b?**Z**
2. ? provides basic type checking
3. b?**Z** type checks b as int
4. *d*?**L** checks that *d* is an l-value

Notational conventions, cont’d

C. "$\bullet \to \bullet , \bullet$" is the if-then-else expr

D. "$\bullet [\bullet \mapsto \bullet]$" is "function perturbation". E.g.,

$$s[I \mapsto r]$$

means

"enter *r* as value of *I* in alist *s*".

VIII. Tennent Section 13.2

A. Language very similar Lisp subset handled by `xeval`

B. Semantic domains:

1. **T** and **Z** are *booleans* and *ints*.

2. **B** is product of bools and ints, called *basic values*.

Tennent 13.2, cont'd

3. **S** is the *store*, as a function from text id's to storable values; think of it as an alist:

Text Id	Basic Value
. . .	

Tennent 13.2, cont’d

4. **P** is the domain of *procedures*.

5. **R** is *storable values*, union of basic vals with procedure vals

6. **E**, **G**, and **A** are **R**, **S**, and **B** resp., with {*error*} added.

IX. Adding an environment (13.3)

A. Language very similar to Lisp subset handled by `xcheck` as well as SIL.

B. A few notational abnormalities:

Tennent	Normal Pascalese
new I = E	**var** Id := Expr
val I = E	**const** Id = Expr
with D **do** C	Decls **begin** Commands **end**

Tennent 13.3, cont'd

C. Notational conventions

1. Add to 13.2 an *environment*, in conjunction with the store:

Environment

Text Id	**Value**

Store

L-Value	**Storable Value**

Tennent 13.3, cont'd

2. We've separated storable and denotable values.

3. More accurately models store as computer memory.

    a. Not done in SIL def.

    b. Could easily be done with attr grammar.

Tennent 13.3, cont'd

4. Can represent semantics of Pascal first-order proc bodies.

5. Interesting to consider semantics of C "&".

6. Adding **L** to RHS of **R** def

   $$r \in \mathbf{R} = \mathbf{B} + \mathbf{P} + \mathbf{L}$$

   defines important aspect of C.

7. Nice illustration of power of denotational semantics.

X. Semantic functions 13.2 & 13.3

A. The meat of the matter.

B. Summary:

Descrip	**13.2**	**13.3**
Expr Eval	$\mathcal{E}$: Exp $\rightarrow$ (S $\rightarrow$ E)	$\mathcal{E}$:E xp $\rightarrow$ (U $\rightarrow$ (S $\rightarrow$ E))
Cmd Exec	$\mathcal{C}$: Com $\rightarrow$ (S $\rightarrow$ G)	$\mathcal{C}$:C om $\rightarrow$ U $\rightarrow$ S $\rightarrow$ G
Decl Elab	---	$\mathcal{D}$: Def $\rightarrow$ U $\rightarrow$ S $\rightarrow$ (U $\times$ G)
Pgm Exec	$\mathcal{M}$: Pro $\rightarrow$ B $\rightarrow$ A	$\mathcal{M}$: Pro $\rightarrow$ B $\rightarrow$ A

XI. Whither inheritance and synthesis?

A. Inherited attributes

1. Args passed *in* to semantic functions.

2. E.g., env and store passed down from $\mathcal{C}$ to $\mathcal{E}$

Inheritance and synthesis, cont'd

B. Synthesized attributes

1. Results passed *out* from semantic functions.

2. E.g., result produced by $\mathcal{E}$ synthesized up to call from $\mathcal{C}$.

3. Similarly, store from $\mathcal{C}$ synthesized up to caller.

XII. Pervasive use of functions.

A. Table-valued alists represented as functions.

B. E.g., both the env and store.

1. `assoc` function replaced by applying function to an ident.

2. Will take some getting used to.