# CSC 530 Lecture Notes Week 6
## *-- REVISED --*
## Discussion of Assignment 3, Questions 1 and 2
## Introduction to Denotational Semantics

I. Turingol Highlights

A. Knuth's Turingol semantics defines the compilation of a simple textual Turing Machine language into lower-level TM quintuples.

B. In comparison to the SIL attribute grammar semantics, the Turingol semantics are *compiled* whereas the SIL semantics are *interpreted*. (An example of a compiled semantics for a SIL-like language is attached to the end of these notes.)

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

< p , A , c , d , q >

where

p = present state
A = symbol scanned
c = symbol written
d = tape movement direction
q = next state

D. Two major attributes in the Turingol definition are **Symbol** and **label**

1. These are used as *symbol tables*, in a manner similar to similar to how the env and store attributes are used in the SIL definition.
2. Viz., the tables store bindings of identifiers to some form of value.
3. In the case of **Symbol** and **label**, they associate program names with TM-level values.
4. E.g., "**tape alpha is** point, blank, one, zero" would have a **Symbol** attribute like this:

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

5. Similarly, statement labels are bound in a **label** symbol table such as

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

E. Some details of Example 4.1 on page 137.

| **Source string** | **TM quintuple** |
|---|---|
| **tape alpha is** blank, one, zero, point | |
| **print** point | $<q_0, s, ., 0, q_1>$ where s = {B,0,1,.} |
| **goto** carry | $<q_1, s, s, 0, q_4>$ |
| test: **if the tape symbol is** *"one"* | $<q_2, s, s, 0, q_6>$ where s = {B,0,.} |
| **then** | $<q_2, 1, 1, 0, q_3>$ |
| {**print** *"zero"* ; | $<q_3, s, 0, 0, q_4>$ |
| . . . | |
| } | |
| realign: **move right one square** | $<q_7, s, s, 0, q_8>$ |

F. Additional notes
1. Σ must be fully processed and available before any instructions are executed
2. **newsymbol** is Lisp's *gensym* -- it creates a brand new symbol name.
3. The set union operators used in the **define** and **include** auxiliary functions maintain set property; this avoids the problem of having to check for multiply defined entries in the symbol tables.

II. Specifics for Assignment 3, Questions 1 and 2

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

B. For question 2:
1. The idea is to make explicit the attribute dependencies
2. **Label** is the most interesting case (think about the semantics of the declaration and use of labels in a typical programming lang)
3. Note that you need not really understand what Turingol programs do or even much of what the TM code looks like -- rather, the focus here is on the general semantic definition technique.

***Now on to Denotational Semantics***

III. Relevant reading: Papers 17-22, primarily number 20.

IV. Introductory comparison of Knuth-style attribute grammar semantics with Tennent-style denotational semantics.

A. In Knuth, we define attributes and semantic rules, but the rule evaluation strategy is not explicitly specified, i.e., precisely *how* we do the tree walk is a up to us.

B. In contrast, the denotational approach replaces the under-specified tree-walk evaluation scheme with formal function evaluation.

1. Rather than walk through a tree any way we feel like, we are forced to perform what amounts to a depth-first traversal
2. This is imposed by the fact that semantic functional arguments are expressed in terms of syntactic constituents.
3. For the denotational approach, the analog of passing attributes through the tree is carried out by passing function arguments between semantic functions.
4. Multiple evaluation passes are still available in a denotational definition, based on one full-pass semantic function invoking another full-pass function.
5. A significant advantage of the denotational approach is availability of evaluation functions as first-class objects, rather than having evaluation based on a parse tree data structure.
   a. This means in particular that we do not need the less-than-completely formal *`function-ize`* auxiliary function.
   b. Defined function bodies are represented not as attributed parse trees, but as the same form of functions used to define the rest of the language semantics.
6. Also, the rather operational definition of looping required in an attribute grammar definition is replaced with a much more mathematical form, using the concept of *fixpoints*.

C. Further examples to follow next week.

V. Data domains, from Tennent Chapter 3

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

B. As with attribute grammar definitions, domain constructions are used for two purposes:
   1. To define the datatypes on which to build denotational definitions.
   2. To model higher-level data constructors in programming languages.

C. Summary of what domain constructions model:
   1. Product domains are *records*
   2. Sum domains are *unions* (aka, *variant records*).
   3. Function domains can be used to model *arrays* and other forms of *tables*.
   4. Also, function domains are used to model the *value* of a procedure body -- not the value it computes, but the value of its body as an unevaluated function (i.e., a lambda expression).
   5. As in Lisp and other functional languages, recursive domains are used to provide the same capabilities as *pointers*.

VI. The binary numeral example of Tennent Chapter 13

A. Tennent Chapter 13 starts with a simple example for the semantics of binary numerals.
   1. The Knuth paper has a similar example.
   2. To begin our investigation of denotational semantics, we'll compare three semantic approaches to defining binary numerals -- denotational, attribute grammars, and operational.

B. Denotational definition of binary numerals

**Abstract syntax:** N ∈ **Nml** = binary numerals
I ∈ **Int** = binary integers
F ∈ **Frac** = binary fractions

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

**Semantic domain:** Q = real numbers

**Semantic functions:** $\mathcal{N}$: **Nml** → Q
$\mathcal{I}$: **Int** → Q
$\mathcal{F}$: **Frac** → Q

$\mathcal{N}$[[I . F]] = $\mathcal{I}$[[I]] + $\mathcal{F}$[[F]]
$\mathcal{I}$[[I B]] = 2*$\mathcal{I}$[[I]] + $\mathcal{I}$[[B]]
$\mathcal{I}$[[0]] = 0
$\mathcal{I}$[[1]] = 1
$\mathcal{F}$[[B F]] = $\mathcal{F}$[[B]] + $\mathcal{F}$[[F]]/2
$\mathcal{F}$[[0]] = 0
$\mathcal{F}$[[1]] = 1/2

**Test case:** 1101.01
$\mathcal{N}$[[1101.01]] = $\mathcal{I}$[[1101]] + $\mathcal{F}$[[01]]
= (2*$\mathcal{I}$[[110]] + $\mathcal{I}$[[1]]) + ($\mathcal{F}$[[0]] + $\mathcal{F}$[[1]]/2)
= (2*(2*$\mathcal{I}$[[11]] + $\mathcal{I}$[[0]]) + $\mathcal{I}$[[1]]) + ($\mathcal{F}$[[0]] + $\mathcal{F}$[[1]]/2)
= (2*(2*(2*$\mathcal{I}$[[1]] + $\mathcal{I}$[[1]]) + $\mathcal{I}$[[0]]) + $\mathcal{I}$[[1]]) + ($\mathcal{F}$[[0]] + $\mathcal{F}$[[1]]/2)
= (2*(2*(2*1 + 1) + 0) + 1) + (0 + (1/2)/2)

**Another test case:** 101100.001101 = 32 + 8 + 4 + .125 + .0625 + .015625 = 44.203125
$\mathcal{N}$[[101100.001101]] =
= $\mathcal{I}$[[101100]] + $\mathcal{F}$[[001101]]
= (2*$\mathcal{I}$[[10110]] + $\mathcal{I}$[[0]])
+ ($\mathcal{F}$[[0]] + $\mathcal{F}$[[01101]]/2)
= (2*(2*$\mathcal{I}$[[1011]] + $\mathcal{I}$[[0]]) + $\mathcal{I}$[[0]])
+ ($\mathcal{F}$[[0]] + ($\mathcal{F}$[[0]] + $\mathcal{F}$[[1101]]/2)/2)
= (2*(2*(2*$\mathcal{I}$[[101]] + $\mathcal{I}$[[1]]) + $\mathcal{I}$[[0]]) + $\mathcal{I}$[[0]])
+ ($\mathcal{F}$[[0]] + ($\mathcal{F}$[[0]] + ($\mathcal{F}$[[1]] + $\mathcal{F}$[[101]]/2)/2)/2)
= (2*(2*(2*(2*$\mathcal{I}$[[10]] + $\mathcal{I}$[[1]]) + $\mathcal{I}$[[1]]) + $\mathcal{I}$[[0]]) + $\mathcal{I}$[[0]])
+ ($\mathcal{F}$[[0]] + ($\mathcal{F}$[[0]] + ($\mathcal{F}$[[1]] + ($\mathcal{F}$[[1]] + $\mathcal{F}$[[01]]/2)/2)/2)/2)
= (2*(2*(2*(2*(2*$\mathcal{I}$[[1]] + $\mathcal{I}$[[0]]) + $\mathcal{I}$[[1]])+ $\mathcal{I}$[[1]]) + $\mathcal{I}$[[0]]) + $\mathcal{I}$[[0]])
+ ($\mathcal{F}$[[0]] + ($\mathcal{F}$[[0]] + ($\mathcal{F}$[[1]] + ($\mathcal{F}$[[1]] + ($\mathcal{F}$[[0]] + $\mathcal{F}$[[1]]/2)/2)/2)/2)/2)
= (2*(2*(2*(2*(2*1 + 0) + 1)+ 1) + 0) + 0)
+ (0 + (0 + (1/2 + (1/2 + (0 + 1/2/2)/2)/2)/2)/2)
= 44.203125

C. Attribute grammar definition of binary numerals

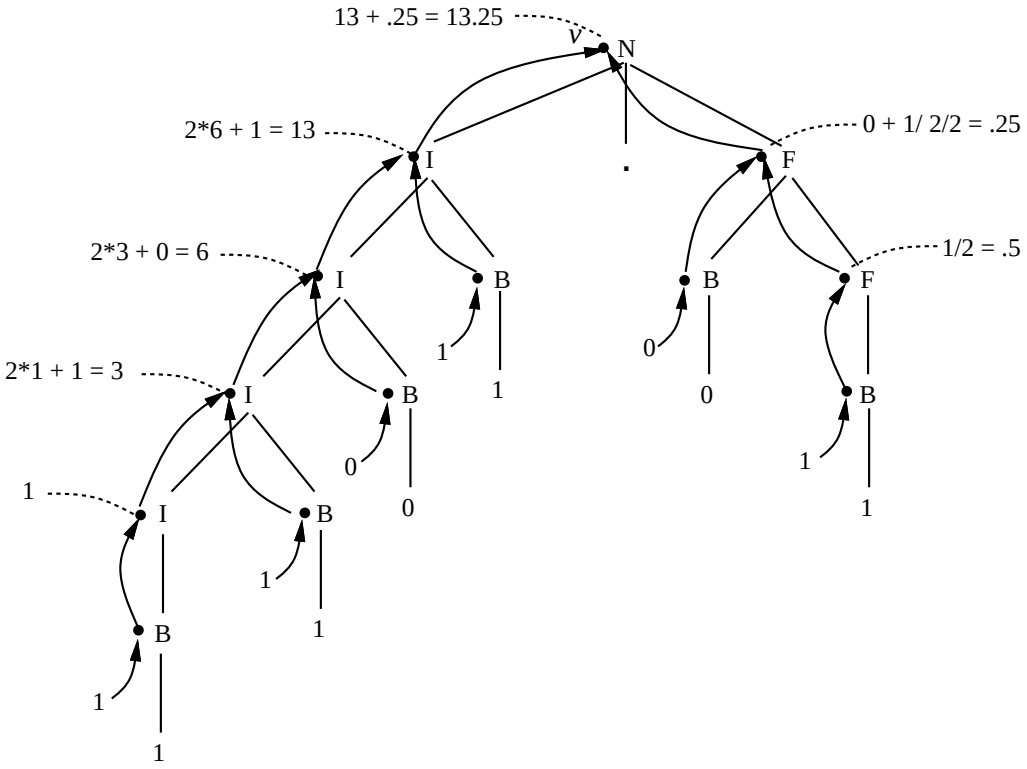
**Attributes:**

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

**Grammar rules 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};
```

**Test case:**




D. Operational definition of binary numerals

```
; 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)))
        ;(break "in mid eval-binary-number")
        (+ integer-value fractional-value)
    )
)

(defun eval-integer-part (number val)
    ;(break "in eval-integer-part")
    (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
;           (print (list "number=" number ", next fp="
;                        (eval-fractional-part (cdr number) val)))
            (let* ((val (/ (eval-fractional-part (cdr number) val) 2.0)))
                (print (list "number=" number ", val=" val))
                (+ (/ (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)) )
    )
)
```

**Test case:**

```
% gcl

>(eval-binary-number '( 1 1 0 1 \. 0 1 ))

13.25
```

E. Some observations on the three techniques.

1. The syntax in the attribute grammar definition is slightly more verbose, or at least more spread out

a. Though it does not show that much in this simple example, the syntax in a denotation definition is generally more abstract than in an attribute definition.
b. To a large extent this is due to the fact that the semantic definitions are directly hung off the grammar rules in an attribute grammar, where as they are factored into a separate list in a denotational definition.
c. What is most important is that both attribute grammars and denotational definition are *syntax-directed*.
d. The difference between abstract versus concrete syntax is really unimportant from a semantics point of view.

2. The heart of the definition in the attribute grammar and denotational semantics are the same.
3. The operational semantics is considerably bulkier than the other two definitions.
   a. A major nuisance in the operational definition is having to deal directly with scanning.
   b. In particular, the function `move-upto-dot` is entirely superfluous to the semantics.
   c. While other approaches to the operational definition could be shorter, the bottom line is that a certain amount of operational hacking must always be done to get an operational definition to work.
   d. In contrast, the syntax-directed foundation of the attribute grammar and denotational techniques obviate the need for any syntactic processing.

VII. Notational conventions for Tennent-style denotational definitions (Chapter 13, pp. 213-216)

A. Double square brackets enclose syntactic operands (i.e., this is an abstraction of all parsing).

B. ? is the union "tag test" operator.
   1. E.g., if *b* is a basic value (bool or int) then b?**T** is true if b is a bool (*T*ruth value) and b?**Z** is true if b is an int (**Z** is the normal math abbreviation for the set of ints).
   2. In terms of environments and stores, ? provides the basic type and class checking mechanism.
   3. I.e,. for a store, the expression b?**Z** type checks b in that it fails if b is not an int.
   4. Similarly for an environment, the expression *d*?**L** checks that *d* is an l-value (see rule for assignment in Table 13.3).

C. "$\bullet \rightarrow \bullet , \bullet$" is the if-then-else expression. E.g.,

$$e \rightarrow x_1, x_2$$

is

**if** *e* **then** $x_1$ **else** $x_2$

D. "$\bullet$ [ $\bullet \mapsto \bullet$ ]" is the "function perturbation", a.k.a, "make an entry in the functional representation of an alist" function. E.g.,

$$s[I \mapsto r]$$

means

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

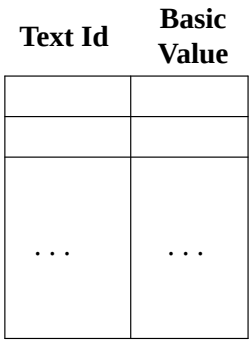
1. Cf. Knuth's **define**, which does the same thing
2. Both the function perturbation operator and Knuth's **define** have the same semantic effect as adding a new binding into an alist.
3. This purely functional view of tables takes some getting used to.

VIII. A Simple Programming Language (Tennent Section 13.2)

A. The denotational semantics of the programming language in Tennent Section 13.2 is very similar to the semantics of the Lisp subset handled by `xeval` in Assignment 1.

B. Notes about the semantic domains:

1. **T** and **Z** are *booleans* and *ints*, i.e., the basic atomic data values. Reals are trivial to add (well, almost trivial).
2. **B** is the product domain (i.e., union) of bools and ints, and hence the set of all *basic values* in this simple language.
3. **S** is the *store*. It is a function from text id's to storable values (more on storable values in just a bit). As such, it can be thought of as an alist of the form:




where the text id's are the keys and the basic values are their corresponding entries.

4. **P** is the domain of *procedures*.
   a. These are unevaluated procedure bodies that are denoted as functions.
   b. The denotation domain of procedures is the analog of the functions generated by the `functionize` auxiliary function used in the attribute grammar definition of SIL and the lambda forms used as the value of `xdefun`'d functions in the operational semantics.
   c. See Tennent bottom of page 219.
5. **R** is the set of *storable values* that is the product of basic values (i.e., the bool and int scalars) together with the procedure values.
   a. Hence, a store entry is either an atomic bool or int, or a function which denotes a proc body.
   b. This is analogous to the two forms of bindings -- values and functions -- in the Lisp interpreter and the SIL attribute semantics.
6. **E**, **G**, and **A** are just **R**, **S**, and **B** respectively with {*error*} tacked on.

IX. Adding an environment to the simple PL (Tennent Section 13.3)

A. The denotational semantics of the programming language in Tennent Section 13.3 is very similar to the semantics of the Lisp subset handled by `xcheck` and `xeval` in Assignment 2, as well as to the semantics of SIL.

B. Reconciling a few notational abnormalities:

| Tennent | Normal Pascalese | What it is |
|---|---|---|
| **new** I = E | **var** ident : type := expr | scoped var decl with initialization |
| **val** I = E | **const** ident = expr | scoped constant decl |
| **with** D **do** C | decls **begin** stmts **end** | block with decls and stmts |

where type is restricted to `integer` or `boolean` and in the case of the **new** declaration, type is inferred from the type of the expression E.

C. Notational conventions

1. Add to definition 13.2 a new *environment* alist to exist in conjunction with the store:

*Environment*

| Text Id | Value |
|---|---|
| | |
| | |
| . . . | . . . |
| | |

*Store*

| L-Value | Storable Value |
|---|---|
| | |
| | |
| . . . | . . . |
| | |

2. In so doing, we've now separated the classes of storable and denotable values.
3. This separation more accurately models the semantics of a store as a raw piece of computer memory, since an l-value models a memory address explicitly.
   a. This separation is not done in the attribute grammar definition of SIL, where the definition of a store is more abstract.
   b. The separation could easily be done using attribute grammars (i.e., it's not something that requires any special feature of the denotational technique).
4. Among other things, we can now represent the proper semantics of Pascal procedure bodies by making elements of the domain **P** denotable but not storable; this reflects the first-order Pascal semantics that does not have procedure types.
5. In this regard, it is interesting to consider the semantics of the C "&" operator -- it makes all elements of the domain **L** *storable* as well as denotable.
6. Hence, with the simple addition of **L** to the RHS of the storable values domain definition:

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

   we can define one of the more important aspects of C semantics that distinguishes it from most other languages
7. This is a pretty nice illustration of the power of denotational semantics.
8. For complete truth in advertising, we should note that the preceding definition is a bit of a simplification, since Pascal allows a restricted class of L-values to be storable, i.e., pointer values.
   a. Therefore, the genuine distinction between Pascal and C L-value semantics would involve subdividing the **L** domain some more.
   b. Nevertheless, this is still a good example of the power -- if not the *density* -- of the denotational technique.

X. The semantic functions of Tennent used in definitions 13.2 & 13.3

A. The meat of the matter in a denotational definition are the semantic functions that deliver the meaning.

B. Here is a summary of the functions used in Tennent definitions 13.2 and 13.3:

| **Description** | **13.2** | **13.3** |
|---|---|---|
| Expression Evaluation | $\mathcal{E}$ : Exp $\rightarrow$ ( S $\rightarrow$ E ) | $\mathcal{E}$ : Exp $\rightarrow$ ( U $\rightarrow$ ( S $\rightarrow$ E )) † |
| Command Execution | $\mathcal{C}$ : Com $\rightarrow$ ( S $\rightarrow$ G ) | $\mathcal{C}$ : Com $\rightarrow$ U $\rightarrow$ S $\rightarrow$ G |
| Declaration Elaboration | --- | $\mathcal{D}$ : Def $\rightarrow$ U $\rightarrow$ S $\rightarrow$ ( U $\times$ G ) |
| Program Execution | $\mathcal{M}$ : Pro $\rightarrow$ B $\rightarrow$ A | $\mathcal{M}$ : Pro $\rightarrow$ B $\rightarrow$ A |

† right associativity shown explicitly for $\mathcal{E}$; same for others

XI. Whither inheritance and synthesis in Tennent-style defs?

A. Inherited attributes

1. In a Tennent-style definition, inherited attributes are represented as arguments passed *in* to semantic functions.
2. E.g., as the env and store are passed down from $\mathcal{C}$ to $\mathcal{E}$, these can be considered inherited by $\mathcal{E}$.

B. Synthesized attributes

1. In a Tennent-style definition, synthesized attributes are represented as function results passed *out* from semantic functions.
2. E.g., the expression value result produced by $\mathcal{E}$ is synthesized up to $\mathcal{C}$ when $\mathcal{C}$ calls $\mathcal{E}$.
3. Similarly, the store value produced by the call*ed* invocation of $\mathcal{C}$ is synthesized up to the call*ing* invocation of $\mathcal{C}$.

XII. The pervasive use of functions as data in Tennent-style defs

A. Many places in a Knuth-style definition where a table-valued (i.e., alist) attribute is used, the comparable attribute in the Tennent-style definition is the function version of that table.

B. E.g., both the env and store in 13.3 are *functions* instead of tables.

1. This means that, for example, the env `assoc` aux function in the Knuth-style SIL semantics is replaced by applying the env function itself to an identifier to yield the table value corresponding to that identifier.
2. As noted earlier, this will take some getting used to.

XIII. Next week we'll dissect definitions 13.2 and 13.3 -- please bring your copy of paper 24 to class.

XIV. Attached is the sample compiler-oriented attribute semantics refered to on page 1 of the notes.

A. You do not need to read through the example in detail.

B. The point of providing the example is to give you a general feel for what code generation semantics look like compared to interpretive semantics.

```
/*
 * This is a Yacc-style attribute grammar for the code generation semantics of
 * a language very similar to SIL.  The meta-notation used here is slightly
 * less formal that the pure ML notation used in the interpretive SIL
 * definition.  The point of this example is to give you a general feel for
 * what code generation semantics look like for a language of other than the
 * very simple Turingol defined by Knuth.
 *
 * The semantic rules define both type checking and code generation in the same
 * definition.  These are the attributes that are used:
 *
 *      NAME    DESCRIPTION
 *      ========================================================================
 *      symtab  A global reference-valued attribute representing an abstract
 *              symbol table.  A symtab is a reference to a 2-tuple of the
 *              form:
 *                  (parenttab, entries)
 *              where parenttab is a reference to the parent symbol table, and
 *              entries is a list of 5-tuples of the form:
 *                  [ (name, class, type, level, offset), ... ].
 *              For symbols with class = "proc", a symtab entry is a nine-tuple
 *              of the form
 *                (name, "proc", type, level, offset, parms, symtab, label, size)
 *              where the first 5 items are as for a variable entry, and the
 *              last 4 items are, respectively, the list of formal parm names,
 *              the local procedure symbol table, the object code label for the
 *              proc, and the size in bytes of the proc act. record.
 *
 *      type    A string-valued attribute representing the names of types.
 *              Successful type checking is represented by the value of
 *              program.type = "OK".  Note that a string-valued type attribute
 *              is used for this simple language since there are no structured
 *              types.  For a language like Modula-2, the type attribute would
 *              be tuple-valued, in order to represented structured types
 *              effectively.
 *
 *      class   A string-valued attribute representing the names of symbol
 *              classes, specifically "var", "parm", or "proc".
 *
 *      text    A string-valued attribute representing the lexical text of
 *              declared identifiers.
 *
 *      types   A sequence-valued attribute that holds a list of types for
 *              formal and actual procedure parameters.
 *
 *      code    A synthesized sequence-valued attribute that holds obj code.
 *
 *      addr    A synthesized string-valued attribute for a machine address.
 *
 *      parms   A sequence-valued attribute that holds the names of formal
 *              procedure parameters.
 *
 *      actuals  A sequence-valued attribute that holds the types of actual
```

```
*               procedure parameters from a proc call.
*
*       codes   Sequence of actual parm code values.
*
*       addrs   Sequence of actual parm addr values.
*
*       label   A global integer attribute used to generate unique labels.
*               Initial value is 0.
*
*       reg     A global integer attribute used to generate an available
*               register.  Initial value is 1.
*
*       curoffset  An inherited integer attr that records the next available
*               storage offset, in bytes.
*
*       level   An inherited integer attribute that records the lexical
*               nesting level.
*
*       size    A synthesized integer attr that records total size in bytes
*               of allocated storage.
*
*       depth   Synthesized integer attribute used to record max nesting depth
*               of proc decls.
*
*       WORDSIZE  A global constant integer attribute that holds the size of a
*               word in bytes.
*
*
* There are three auxiliary functions used to enter and lookup symbols:
*
*       Enter(symtab, symbol, class, type, level, offset) =
*              let (!symtab) 2 =
*                    (!symtab) 2 U (symbol, class, type, level, offset)
*
*       Enter(symtab, symbol, class, type, level, offset, parms, symtab)
*              let (!symtab) 2 = (!symtab) 2 U
*                    (symbol, class, type, level, offset, parms, symtab)
*
*       Lookup(symtab, symbol) = the first element S of (!symtab) 2
*                                       such that S 1 = symbol,
*                                "ERROR" if no such element
*
*
* There are three auxiliary functions used to enter and exit procedure scopes
* while symtab values are being computed:
*
*       EnterProc(symtab, name, level) =
*               let newsymtab = (symtab, {})
*               Enter(symtab, name, "proc", "", level, 0, (), newsymtab)
*               let Lookup(name) 8 = NextLab()
*               let symtab = newsymtab
*
*       EnterParms(symtab, name, type, parmtypes, offset) =
```

```
*              let Lookup(symtab, name) 3 = type
*              let Lookup(symtab, name) 4 = parmtypes
*              let Lookup(symtab, name) 5 = offset;
*              let Lookup(symtab, name) 9 = offset + 2 * WORDSIZE
*
*      ExitProc(symtab) =
*              let symtab = symtab 1
*
*
* There is an aux function to compute the size in bytes of a data type.  With
* the simple types of the Translator 5 language, this is a simple function.
* For Modula-2, it's more complicated, as has been noted before.
*
*      typesize(t) =
*              if t = "integer" then WORDSIZE
*              else if t = "real" then WORDSIZE
*              else if t = "boolean" then 1
*              else if t = "char" then 1
*
*
* There are aux functions used to generate unique labels and available
* registers, assuming that initially label has value 0 and reg has value 1.
*
*      NextLab() = "L" || label; let label = label + 1
*      NextReg() = "R" || reg; let reg = reg + 1
*      ClearRegs() = let reg = 1
*
* There are two aux functions to generate the largely canned pieces of code
* that go at the top of the object code, and at the top of the main body:
*
*      topcode(size,depth) =
*              ["GOTOMAIN",
*               "STATIC\tDATA\t" || strify(size),
*               "STACK\tDATA\t25000",
*               "DISPLAY\tDATA\t" || strify(depth*WORDSIZE)]
*
*      maincode() =
*              ["MAIN\tDATA\t0",
*               "\tMOV\tSTATIC, R0",
*               "\tMOV\tSTACK, SP",
*               "\tADD\t 25000, SP",
*               "\tMOV\tDISPLAY, R1"]
*
*
* There is an aux function to generate a local or global machine address:
*
*      genaddr(sym) =
*              if sym 4 = 0
*              then strify(sym 5) || "(R0)"
*              else strify(sym 5) || "(SP)");
*
*
* There is a simple aux function to generate an assignment stmt:
```

```
 *
 *      genassmnt(src, dest) = "\tMOV\t" || src || ", " || dest
 *
 *
 * There are two simple aux functions to gen code to push and pop act records:
 *
 *      pushactrec(size) =
 *              "\tSUB\t" || strify(size) || ", SP"
 *      popactrec(size) =
 *              "\tADD\t" || strify(size) || ", SP"
 *
 *
 * Finally, there is an aux function to generate code for the details of a proc
 * call:
 *
 *      gencall(proclab, returnlab, offset) =
 *              ["\tMOV\t" || returnlab || ", " || strify(offset) || "(SP)",
 *               "\tGOTO\t" || proclab,
 *               returnlab || "\tDATA\t0"]
 *
 *
 */

program          : YPROGRAM decls YBEGIN stmts YEND
                        {symtab =
                            [ ref (), [("integer", "type", "integer"), ...] ];
                         $$.type = $4.type;
                         $2.level = 0;
                         $2.curoffset = 0;
                         if ($$.type = "OK" then
                            $$.code = topcode($2.size, $2.depth) U
                                $2.code U maincode() U $4.code;}
                ;

decls           : /* empty */
                        {$$.size = 0;
                         $$.depth = 0;
                         $$.code = [];}
                | decl ';' decls
                        {$1.level = $$.level;
                         $3.level = $$.level;
                         $1.curoffset = $$.curoffset;
                         $3.curoffset = $$.curoffset + $1.size;
                         $$.size = $1.size + $3.size;
                         $$.depth = max($1.depth, $3.depth);
                         $$.code = $1.code U $3.code;
                        }
                ;

decl            : vardecl
                        {$1.level = $$.level;
                         $1.curoffset = $$.curoffset;
                         $$.size = $1.size;
```

```
                        $$.depth = $1.depth;
                        $$.code = [];
                       }
                | procdecl
                       {$1.level = $$.level;
                        $$.size = 0;
                        $$.depth = $1.depth;
                        $$.code = $1.code;
                       }
                ;

vardecl         : YVAR vars ':' type
                       {$2.type = $4.type; /* notice inherited attribute */
                        $2.class = "var";
                        $2.level = $$.level;
                        $2.curoffset = $$.curoffset;
                        $$.size = $2.size;
                        $$.depth = 0;
                       }
                ;

type            : Yidentifier
                       {$$.type = Lookup(symtab, $1.text) 3;}
                ;

vars            : var
                       {Enter(symtab, $1.text, $$.class, $$.type,
                              $$.level, $$.curoffset);
                        $$.size = typesize($$.type)}
                | var ',' vars
                       {$3.type = $$.type;
                        $3.class = $$.class;
                        $3.level = $$.level;
                        Enter(symtab, $1.text, $$.class, $$.type,
                              $$.level, $$.curoffset);
                        $3.curoffset = $$.curoffset + typesize($$.type);
                        $$.size = typesize($$.type) + $3.size;
                       }
                ;

var             : Yidentifier
                       {$$.text = $1.text;
                          /* The lexer provides Yidentifier as a string */
                        let sym = Lookup(symtab, $1.text);
                       }
                ;

procdecl        : YPROCEDURE prochdr ';' procbody
                       {$2.level = $$.level;
                        $4.level = $$.level + 1;
                        $4.curoffset = $2.size;
                        $$.depth = $4.depth;
                        $$.size = $4.size;
```

```
                        $$.code = $2.code || $4.code;
                        ExitProc($2.text);}
                       }
                ;

prochdr         : Yidentifier '(' formals ')'
                       {EnterProc(symtab, $1.text, $$.level);
                        $3.level = $$.level + 1;
                        $3.curoffset = 0;
                        EnterParms(symtab, $1.text, "",
                               $3.parms, $3.size);
                        $$.text = $1.text;
                        $$.code = Lookup(symtab, $1.text) 7 || "\tDATA\t0";}
                       }
                | Yidentifier '(' formals ')' ':' type
                       {EnterProc(symtab, $1.text, $$.level);
                        $3.level = $$.level + 1;
                        $3.curoffset = 0;
                        EnterParms(symtab, $1.text, $6.type,
                               $3.parms, $3.size);
                        $$.text = $1.text;
                        $$.code = Lookup(symtab, $1.text) 7 || "\tDATA\t0";}
                       }
                ;

formals         : /* empty */
                       {$$.types = null;
                        $$.parms = null;}
                | formal
                       {$$.types = $1.type;
                        $1.level = $$.level;
                        $1.curoffset = $$.curoffset;
                        $$.parms = $1.text;
                        $$.size = $1.size;
                       }
                | formal ',' formals
                       {$$.types = $1.type U $3.types;}
                        $$.types = null;
                        $1.level = $$.level;
                        $1.curoffset = $$.curoffset;
                        $3.curoffset = $$.curoffset + $1.size;
                        $$.parms = $1.text U $3.parms;
                        $$.size = $1.size + $3.size;
                       }
                ;

formal          : Yidentifier ':' type
                       {Enter(symtab, $1.text, "parm", $3.type,
                               $$.level, $$.curoffset);
                        $$.text = $1.text
                        $$.type = $3;
                        $$.size = typesize($3.type);
                       }
```

```
                ;

procbody        : decls YBEGIN stmts YEND
                       {$$.type = $3.type;
                        $1.level = $$.level;
                        $1.curoffset = $$.curoffset;
                        $$.size = $1.size;
                        $$.depth = $1.depth + 1;
                        $$.code = $3.code;
                       }
                ;

stmts           : stmt ';'
                       {$$.type = $1.type;
                        $$.code = $1.code;}
                | stmt ';' stmts
                       {$$.type = if ($1.type = "OK" and $3.type = "OK")
                                       then "OK" else "ERROR";
                        $$.code = $1.code U $4.code;
                       }
                ;

stmt            : /* empty */
                | Yidentifier YASSMNT expr
                       {$$.type =
                            (if Lookup(symtab, $1.text) 3 = $3.type
                             then "OK" else "ERROR");
                        $$.code =
                             $3.code U genassmnt($3.addr, $1.addr)
                        ClearRegs();
                       }
                | Yidentifier '(' actuals ')'
                       {let ftypes = Lookup(symtab, $1.text) 4;
                        let proctype = Lookup(symtab, $1.text) 3;
                        $$.type =
                            if (foreach (fp in ftypes, ap in $3.types)
                                (fp = ap) and (proctype = null))
                            then
                                "OK"
                            else
                               "ERROR";
                        $$.code =
                            /* Gen code for proc call stmt, ... */
                        ClearRegs();
                       }
                | YIF expr YTHEN stmts YEND
                       {$$.type =
                            if ($2.type = "boolean") and
                               ($4.type = "OK")
                            then "OK"
                            else "ERROR";
                        $$.code  = /* filled-in code template for if */
                        ClearRegs();
```

```
                    }
                | YIF expr YTHEN stmts YELSE stmts YEND
                    {$$.type =
                         if ($2.type = "boolean") and
                            ($4.type = "OK") and
                            ($6.type = "OK")
                         then "OK"
                         else "ERROR";
                     $$.code  = /* filled-in template for if-then-else */
                     ClearRegs();
                    }
                ;

expr            : number
                    {$$.type = $1.type;
                     $$.code = [];
                     $$.addr = $1.addr;}
                | char
                    {$$.type = $1.type;
                     $$.code = [];
                     $$.addr = $1.addr;}
                | bool
                    {$$.type = $1.type;
                     $$.code = [];
                     $$.addr = $1.addr;}
                | Yidentifier
                    {$$.type = Lookup(symtab, $1.text) 3;
                     $$.code = [];
                     $$.addr = genaddr(Lookup(symtab, $1.text));
                    }
                | Yidentifier '(' actuals ')'
                    {let ftypes = Lookup(symtab, $1.text) 4;
                     let proctype = Lookup(symtab, $1.text) 3;
                     $$.type =
                         if (foreach (fp in ftypes, ap in $3.types)
                             (fp = ap) and (proctype != null))
                         then
                             proctype
                         else
                            "ERROR";
                       /* Gen code as for proc call stmt, plus set
                        * $$.addr = machine addr of return value. */
                    }
                | expr relop expr      %prec '<'
                    {$$.type =
                        if ($1.type = $3.type)
                          and (($1.type = "real") or ($1.type = "integer")
                               or ($1.type = "char"))
                        then $1.type
                        else "ERROR";
                    }
                    /* NOTE: relop code gen not done here. */
                | expr addop expr      %prec '+'
```

```
                        {$$.type =    /* For simplicity, this is Mod-2 rule */
                            if ($1.type = $3.type)
                              and (($1.type = "real") or ($1.type = "integer"))
                            then $1.type
                            else "ERROR";
                             $$.addr =
                                 if isreg($1.addr) then $1.addr else NextReg();
                             $$.code =
                                 if not isreg($1.addr)
                                 then
                                      "\tMOV\t" || $1.addr || ", " || $$.addr  U
                                      "\t" || $2.code || "\t" ||
                                            $3.addr || "," || $$.addr
                                 else
                                      "\t" || $2.code || "\t" ||
                                            $3.addr || "," || $$.addr;
                        }
                | expr multop expr     %prec '*'
                        {$$.type =
                            if ($1.type = $3.type)
                              and (($1.type = "real") or ($1.type = "integer"))
                            then $1.type
                            else "ERROR";
                        }
                        {$$.code = /* ... as for addop */;}
                | '(' expr ')'
                        {$$.type = $2.type;
                         $$.addr = $2.addr;
                         $$.code = $2.code;}
                ;

addop           : '+'
                        {$$.code = "\tADD\t";}        /* etc. for the rest */
                                /* NOTE: this does not handle FADD */
                | '-'
                | YOR
                ;

multop          : '*'
                | '/'
                | YAND
                ;

relop           : '<'
                | '>'
                | '='
                | YLEOP
                | YGEOP
                | YNEOP
                ;

actuals         : /* empty */
                | actual
```

```
                        {$$.types = $1.type;
                         $$.addrs = $1.addr;
                         $$.codes = $1.code;}
                | actual ’,’ actuals
                        {$$.types = $1.type U $3.types;
                         $$.addrs = $1.addr U $3.addrs;
                         $$.codes = $1.code U $3.codes;
                        }
                ;

actual          : expr
                        {$$.type = $1.type;
                         $$.addr = $1.addr;
                         $$.code = $1.code;
                        }
                ;

number          : real
                        {$$.type = $1.type;
                         $$.addr = $1.addr;
                        }
                | integer
                        {$$.type = $1.type;
                         $$.addr = $1.addr;
                        }
                ;

real            : Yreal
                        {$$.type = "real";
                         $$.addr = " " || strify($1.val);
                        }
                ;

integer         : Yinteger
                        {$$.type = "integer";
                         $$.addr = " " || strify($1.val);
                        }
                ;

char            : Ychar
                        {$$.type = "char";
                         $$.addr = "’" || strify($1.val) "’";
                        }
                ;

bool            : Ybool
                        {$$.type = "boolean";
                         $$.addr = " " || strify($1.val);
                        }
                ;
```