Semantic Prototyping: Implementing Action Semantics in Standard ML

Ruth Ruei

and

Ken Slonneger

The University of Iowa

1993

Table of Contents

1. Introduction

Over the past few years Peter Mosses (with the collaboration of David Watt) has developed a framework, known as action semantics, for formally describing the semantics of programming languages. The goal of their efforts has been to produce formal semantic specifications that directly reflect the ordinary computational concepts of programming languages and that are easy to read and understand.

Action semantics has evolved out of the tradition of denotational semantics, where syntactic entities (abstract syntax trees) are mapped compositionally by semantic functions into semantic entities that act as the denotations of the syntactic objects. The chief difference between the two methods of formal specification lies in the nature of the semantic entities. The semantic functions of denotational semantics map syntactic phrases into primitive mathematical values, structured objects, and such higher-order functions as are found in the lambda calculus where functions can be applied to other functions. In contrast, action semantics uses three kinds of first-order entities as denotations: **actions**, **data**, and **yielders**. "First-order" means that actions cannot be applied to other actions.

- The semantic entities known as **actions** incorporate the performance of computational behavior, using values passed to them to generate new values that reflect changes in the state of the computation.
- The **data** entities consist of mathematical values, such as integers, Boolean values, and abstract cells representing memory locations, that embody particles of information. Sorts of data used by action semantics are defined by algebraic specifications (see the references).
- **Yielders** encompass unevaluated pieces of data whose values depend on the current information incorporating the state of the computation.

In action semantics, the semantics of a programming language is defined by mapping program phrases to actions. The performance of these actions relates closely to the execution of the program phrases. Primitive actions can store data in storage cells, bind identifiers to data, compute values, test truth values, and so on. For example, the following primitive actions contribute to describing programming languages:

complete	Terminate normally the action being performed.
fail	Abort the action being performed.
give _	Give the value obtained by evaluating a yielder.

allocateacell	Allocate a memory location.
store _ in _	Store a value in a memory location.
bind _ to _	Bind an identifier to data produced by a yielder.

These examples illustrate a syntactic convention wherein parameters to operations are indicated by underscores. Operations in action semantics can be prefix, infix, or outfix. Outfix operators have only internal place holders such as in “sum'(_,_)”. The last two examples above are considered prefix since they end with a place holder. Infix operators begin and end with argument places. The operations are evaluated with prefix having the highest precedence and outfix the lowest. Prefix operators are executed from right to left, and infix from left to right.

Other operations, the yielders in action semantics, simply give values that depend on the current information, such as the current storage and the current bindings:

the _ stored in _	Yield the value of a certain type stored in a memory location.
the _ bound to _	Yield the object of a certain type bound to an identifier.

For these yielders, the first parameter specifies the sort of data expected from the operations. Action combinators combine existing actions, normally using infix notation, to control the order in which subactions are performed as well as the data flow to and from their subactions. Action combinators are used to define sequential, selective, iterative, and block structuring control flow as well as to manage the flow of information between actions. The following two combinators model sequential control and nondeterministic choice, respectively.

_ then _	Perform the first action; when it completes, perform the second action.
_ or _	Perform either one of the two actions, choosing one arbitrarily; if it fails, perform the other action using the original state.

Unless otherwise specified (for example, with the combinator or), primitive actions, yielders, and action combinators complete (terminate normally) when the computations they depend on (their parameters) do not fail. Failure at any point in the performance of an action normally propagates to enclosing operations.

With these few operations, we can specify several programming language constructs:

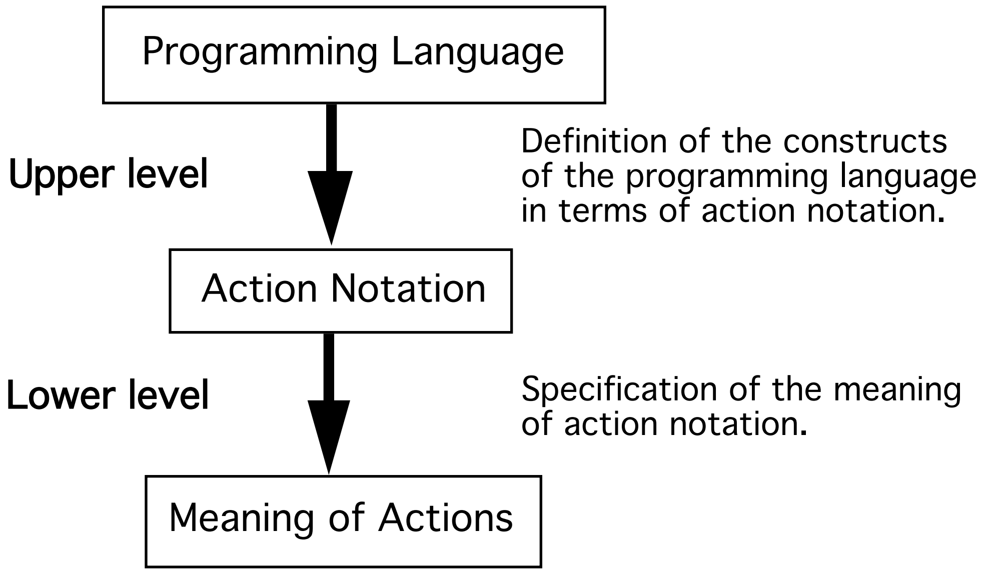
```
elaborate ⟦var I : T⟧ =
                allocateacell
          then
                bind I to the given Cell

elaborate ⟦const I ~ E⟧ =
                evaluate E
          then
                bind I to the given Value

evaluate ⟦I⟧ =
                give the Value stored in the Cell bound to I
          or
                give the Value bound to I
```

These examples are slightly simpler than the semantics of the programming language considered here but convey the basic idea of action specifications. One of the actions in the last example must fail, thereby producing either the constant binding or the variable binding to the identifier. Notice that these semantic definitions can be understood at a superficial level even without a deep knowledge of action semantics. In the sequel we describe these primitive action and action combinators in more detail.

A specification of a programming language using action semantics naturally breaks into two parts:


Programming Language
Upper level
Definition of the constructs of the programming language in terms of action notation.
Action Notation
Lower level
Specification of the meaning of action notation.
Meaning of Actions

The description of action semantics in [Mosses92] specifies the meaning of action notation formally using algebraic axioms to present the notation for data, yielders, and actions, and structural operational semantics to give the semantics of action performance. In this report we describe action notation informally and provide a semantics of action performance by presenting an implementation of action notation in Standard ML.

Using this lower level description of action performance, we present a prototype implementation of an imperative programming language Δ, called Triangle and defined by David Watt, by translating the upper level specification of Δ in action semantics into ML. The implementation closely follows the semantic prototype proposed by Watt in his book *Programming Language Syntax and Semantics* [Watt91]. The BNF syntax and the abstract syntax of Δ are given in Appendix A and Appendix B, respectively.

Section 2 presents an implementation of action semantics in ML (the lower level specification), and section 3 gives an action semantic definition of the programming language Δ (the upper level). Section 4 describes how to run the prototype implementation of Δ. Action semantics itself and the programming language Δ will be introduced gradually as the text progresses.

2. Implementation of Action Semantics in ML

In this section, the three semantic entities, data, yielders, and actions, as well as the concept of sorts are discussed. Actions are the engines that process data and yielders. Yielders are entities that, depending on the current information (the current state of the computation), can be evaluated to yield data. Data are classified into sorts in action semantics. In addition, we describe how the current information is organized in action semantics.

The main features of action notation are developed independently of a particular language specification as much as possible. However, to illustrate the purpose of various parts of action notation, the examples are put in the context of the programming language Δ, which will be specified more fully in section 3.

2.1 Data

Data are pieces of information that are processed by actions in the course of computation. All the distinct sorts of data manipulated by a programming language need to be defined in its action semantics description.

The programming language Δ employs three simple data types, namely **truth values**, **integers** and **characters** and two compound data types, **arrays** and **records**. Variables, representing locations in memory, constitute the essence of an imperative programming language and therefore are included as a sort of data. Storage locations are modeled as **cells** for simple data types and **array variables** and **record variables** for the two compound data types. **Argument lists** are the data structures used to pass data to functions and procedures, while **text**, consisting of a list of characters, is used for input and output. **Indirections** and **unknown** are two other sorts of data that are needed to manipulate recursive bindings; they will be discussed in detail later. Actions themselves are not data but they can be encapsulated in **abstractions,** which are a sort of data.

The distinct sorts of data in our action semantics description are represented in ML as follows.

```
datatype Datum =
    truthvalue'       of bool
  | integer'          of int
  | char'             of string
  | arrayvalue'       of Datum list              (*  Value list              *)
  | arrayvariable'    of Datum list              (*  Variable list           *)
  | recordvalue'      of (Token * Datum) list    (*  (Token * Value) list    *)
  | recordvariable'   of (Token * Datum) list    (*  (Token * Variable) list *)
  | argumentlist'     of Datum list              (*  Argument list           *)
  | text'             of Datum list              (*  Char list               *)
  | cell'             of int                     (*  integers for locations  *)
  | indirection'      of int
  | unknown'
  | abstraction'      of Action * Transients * Bindings;
```

The definition of Datum presupposes the specification of Transients, Bindings, and Action. They will be described later. Note that abstractions are defined as triples consisting of an action, together with transients and bindings. Abstractions are used to model procedural objects in a programming language. The type Token will be defined later to be of type string to give a representation for identifiers.

2.2 Sorts

In action semantics, data are classified into sorts that are equipped with various operations over the elements in the sorts. As with any semantic methodology, programs are expected to be syntactically correct, including both an adherence to the context-free syntax (BNF) and the context-sensitive syntax (context constraints dealing with type checking) before they are submitted to semantic analysis. In spite of this, action semantics follows a strict type discipline in specifying the meaning of language constructs. This careful delineation of types of objects manipulated by actions adds to the information conveyed by the semantic descriptions. Performing an action corresponding to a language construct that violates type constraints results in failure.

In this implementation, each sort is represented by a function of the following type:

```
type Sort = Datum -> Datum;
```

When a value of a particular sort is expected, the appropriate Sort function verifies that the object in question belongs to the sort.

Simple Sorts

Various sorts of data serve as parameters to actions that describe type conformity using the sort information. Each Sort function is defined as an identity function if its argument datum is an element of the represented sort; otherwise it raises an exception. For example,

```
exception Failure;

val TruthValue : Sort =
        fn (truthvalue' tr)     => truthvalue' tr
         | _                    => raise Failure;
```

Sorts for integers, characters, array values, array variables, record values, record variables, argument lists, text, cells, indirections, and abstractions are defined in a similar way.

The ML base types bool, int and string implement the simple data types TruthValue, Integer, and Char, respectively. Note that in Δ an item of type Char is a single character. Therefore, a size check is included in the function for Sort Char.

```
val Char : Sort =
        fn (char' ch)    => (case (size ch) of
                                    1    => char' ch
                                  | _    => raise Failure)
         | _             => raise Failure;
```

Another example tests whether an object is an abstraction, which contains three kinds of entities, an action, transients, and bindings.

```
val Abstraction : Sort =
        fn (abstraction' (A, ts, bs))   => abstraction' (A, ts, bs)
         | _                            => raise Failure;
```

An abstraction encapsulates an action together with the transients and bindings that will be used when the action is eventually performed as a result of the action enact, which takes an abstraction as its parameter and executes the enclosed action using the associated transients and bindings.

Compound Sorts

These sorts are formed as the union of other sorts. The infix operation “\” implements the sort join operation. Action semantics normally uses “|” for the join operation.

```
infix 6 \;

fun (S1:Sort) \ (S2:Sort) =
        fn (d: Datum) =>   S1 d
                           handle Failure => S2 d;
```

A programming language can be categorized by the kinds of values that are produced by expressions, bound to identifiers, stored in memory, and passed as arguments in the language. An action semantics specification requires a sort definition for each of these kinds of data.

Value

The sort **Value**, a subsort of Datum, corresponds to the expressible values in denotational semantics. In Δ the sort Value includes truth values, integers, characters, array values, and record values.

```
val Value : Sort = TruthValue \ Integer \ Char \ ArrayValue \ RecordValue
```

Bindable

Entities that can be bound to identifiers are called denotable values in denotational semantics, while in action semantics they belong to the sort **Bindable**. The sort Bindable, a subsort of Datum, includes the elements of Value (defined above), together with variables, procedures, functions, allocators and indirections for recursive bindings in Δ. Allocators are procedural abstractions that encapsulate the allocation of a variable of a certain type. These objects are bound to type identifiers as a result of declarations in Δ.

```
val Bindable : Sort = Value \ Variable \ Procedure
                           \ Function \ Allocator \ Indirection
```

Storable

The sort **Storable**, also a subsort of Datum, corresponds to the storable values in denotational semantics; it contains all the entities that can be stored in individual cells. In Δ the sort Storable comprises truth values, integers, characters, and text.

val Storable : Sort = TruthValue \ Integer \ Char \ Text;

Variable

Variables represent memory locations. In Δ the sort **Variable**, another subsort of Datum, incorporates cells, array variables, and record variables.

val Variable : Sort = Cell \ ArrayVariable \ RecordVariable;

Function, Procedure, and Allocator

Functions and procedures are denoted by abstractions in action semantics. Recall that allocators, of sort Bindable, are a special kind of procedure used to manage storage allocation. The sorts **Procedure**, **Function**, and **Allocator** are simply defined to be the same as the sort Abstraction.

val Procedure : Sort = Abstraction;

val Function : Sort = Abstraction;

val Allocator : Sort = Abstraction;

Argument

Programming languages have restrictions on what can be passed as arguments to functions and procedures. In Δ arguments can be any type of values or variables as well as functions and procedures. Sort **Argument**, a subsort of Datum, is defined here to ensure the appropriateness of arguments.

val Argument : Sort = Value \ Variable \ Procedure \ Function;

2.3 Yielders

Before we can define yielders, which depend on the current information consisting of transients, bindings, and storage, we need to discuss these kinds of information.

Transients

Intermediate results passed between actions are called **transients** and take the form of a single datum or a tuple of data. These values model the data given by expressions. They must be used immediately or be lost. Transients are represented as lists of data in the ML implementation, so that a single datum is a singleton list. In fact, a singleton tuple is identified with the single datum it contains in action semantics.

type Transients = Datum list;

The auxiliary function **take** returns the nth datum in the given transients.

```
fun take (ts:Transients, n:int) : Datum =
      case ts of   nil      => raise Failure
                 | h::t     => if n = 1   then h
                                else if n > 1   then take (t, n-1)
                                                else raise Failure;
```

Bindings

Bindings are maps from identifiers to data representing **scoped information,** the kind of information modeled by environments in denotational semantics. Bindings are accessible (visible) throughout the performance of an action and its subactions, although they may be hidden temporarily by the creation of inner scopes. Identifiers are tokens implemented as strings of characters.

```
type Token = string;
```

Therefore, the entity Bindings is implemented as lists of Token-Datum pairs.

```
type Bindings = (Token * Datum) list;
```

Five auxiliary functions are defined to manipulate bindings. A short description and the definition of each function are given below; these ML definitions can also be found under “structure DataEquations1” in Appendix C.

- A constant function **empty** returns the empty binding, which is an empty list.

  ```
  val empty : Bindings = [];
  ```

- An **association** function assoc returns the singleton binding of the given identifier to the given datum.

  ```
  fun assoc (t:Token) (d:Datum) : Bindings = [(t,d)];
  ```

- A **find** function returns the datum bound to the given identifier in the given bindings.

  ```
  fun find (bs:Bindings) (t:Token) : Datum =
        case bs of  nil              => raise Failure
                  | (t', dat)::rest  => if t' = t   then dat
                                                    else find rest t;
  ```

- A **merge** function returns the combination of the two given bindings, raising an exception if identifiers clash (see Appendix C for the definition of merge).

```
exception IdeClash;

fun merge (bs1:Bindings) (bs2:Bindings) : Bindings = ... ;
```

- An **overlay** function returns the combination of the two given bindings, with the bindings given first overriding the second collection of bindings.

```
fun overlay (bs1:Bindings) (bs2:Bindings) : Bindings = bs1 @ bs2;
```

Yielders

Given the transients and the bindings, yielders can be evaluated to yield data. A yielder also depends on the current storage, but in this implementation storage is handled as a global object. Therefore, the type Yielder is represented by a function of the following type.

```
type Yielder = Transients * Bindings -> Datum;
```

A datum can be thought of as a special case of Yielder that yields itself when evaluated. We make a datum into a yielder using the function **yield**.

```
fun yield (d:Datum) : Yielder = fn (ts, bs) => d;
```

Each distinct sort has operations that operate on yielders producing data of the proper sort. For example,

```
fun not' (y:Yielder) : Yielder =
    fn (ts, bs) =>
          let val truthvalue' tr = TruthValue (y (ts, bs))
            in
                truthvalue' (not tr)
          end;
```

The function y (ts, bs) is expected to yield a TruthValue. The function not' returns a TruthValue that is the logical *not* of the given argument when given the current transients and bindings. The logical *and* (both) and *or* (either) functions are also defined for TruthValue.

Similarly, operations on integers model the computation carried out when a program is executed, for example,

```
fun sum (y1:Yielder, y2:Yielder) : Yielder =
    fn (ts, bs) =>
          let val integer' n1 = Integer (y1 (ts, bs))
             and integer' n2 = Integer (y2 (ts, bs))
                in
                      integer' (n1 + n2)
          end;
```

The infix operation is determines whether two yielders produce the same datum.

```
fun (y1:Yielder) is (y2:Yielder) : Yielder = … ;
```

A complete list of operations on the distinct sorts can be found in the ML listing as the module "structure DataEquations1" in Appendix C. Additional operations that produce yielders are associated with the various kinds of current information that actions process. These are discussed with the actions themselves.

Storage and the Redirection Table

There are two permanent structures in this implementation: storage and the redirection table. Both of them can be implemented in the same way as transients and bindings, can be passed as arguments from one action to another, and can be given to yielders. However, we choose to implement them imperatively using ML array structures that are visible throughout the entire ML code to reduce the number of arguments passed between actions.

ML array functions, sub and update, search and update arrays, respectively. The function sub(a, i) returns the value stored as the i^{th} element of array a; update(a, i, x) updates the i^{th} element of array a to be the value x.

Storage

Storage consists of an arbitrary number of cells. A cell in the storage has to be in one of the three states: unused' (not allocated), undefined' (allocated but not having a value assigned to it yet), or containing a storable value. Data stored in cells are classified as **stable information**. Changes in storage made during action performance are enduring, so that stable data may only be altered by explicit actions. The ML operation array produces an array of the given size with origin zero and with each component initialized to the given value.

```
datatype State = unused' | undefined' | stored' of Datum;
val StorageSize = 1000;
val Storage = array (StorageSize, unused');
```

Redirections

The **redirection table** consists of an arbitrary number of indirections. Indirections can be bound to identifiers and are used to specify self- and mutually-referential bindings. Cells are locations in the storage, and indirections are entries in the redirection table. An indirection in the redirection table is either not in use or contains a redirection that is a Bindable or the special value unknown'.

```
datatype RedirectionState = notused' | redirection' of Datum;
val RedirectionSize = 1000;
val Redirections = array (RedirectionSize, notused');
```

In a recursive definition of a subprogram with identifier I, the binding of I to an indirection with value unknown' is included in the bindings (environment) encapsulated with the subprogram. The indirection is then changed to refer to the subprogram itself, thereby establishing a cyclic structure that allows a recursive call of the subprogram.

2.4 Actions

Actions are the operational entities in action semantics. Actions are defined to carry out computational tasks. The point of action semantics is to provide a collection of elementary actions that model the individual computational duties performed when a program is executed.

A performance of an action uses the current information, namely the given transients, the received bindings, and the current state of storage, to give new transients, produce new bindings, and/or update the state of the storage. If no intermediate result is to be passed on to the next action, the transient is simply the empty tuple, in this case, the empty list. Similarly, the empty binding is passed to the next action if the action produces no bindings.

An action performance may **complete** (terminate normally), **fail** (terminate abnormally), or **diverge** (not terminate at all). Actions are defined as functions from a transients-bindings pair to another transients-bindings pair since the storage is a permanent structure in this implementation.

```
type Action = Transients * Bindings -> Transients * Bindings;
```

Action Notation

Depending on the principal type of information processed, actions are classified into different facets, including:

- the **functional facet** that processes transient information,
- the **declarative facet** that processes scoped information,
- the **imperative facet** that processes stable information,
- the **reflective facet** that handles abstractions,

- the **basic facet** that provides a means to specify flow of control, and
- **hybrid action notation** that deals with recursive bindings.

2.4.1 Functional Facet

The primitive action **check** with signature check :: Yielder → Action serves as a guard; it completes returning empty transients and bindings when its argument evaluates to true, otherwise it fails.

```
fun check (tv:Yielder) : Action =
    fn (ts, bs) =>
          case tv (ts, bs) of
                truthvalue' true         => (nil, empty)
              | truthvalue' false        => raise Failure
              | _                        => raise Failure;
```

The primitive action **regive** simply gives the given transients.

```
val regive : Action = fn (ts, bs) => (ts, empty);
```

The primitive action **give** with signature give :: Yielder → Action gives the datum yielded by evaluating its argument.

```
fun give (y:Yielder) : Action =
    fn (ts, bs) =>
          let val dat = y (ts, bs)
           in
                ([dat], empty)
          end;
```

Action combinators have the signature,

combinator :: Action * Action → Action.

The functional action combinator **then'** performs the first action using the transients and the bindings passed to the combined action and then performs the second action using the transients given by the first action and the bindings received by the combined action. The transients given by the combined action are the transients given by the second action; the bindings produced by the combined action are the bindings produced by the first action merged with those produced by the second. Since "then" is a reserved word in ML, we use then' for this combinator.

```
infix 1 then';

fun (A1:Action) then' (A2:Action) : Action =
    fn (ts, bs) =>
          let  val (ts1, bs1) = A1 (ts, bs);
               val (ts2, bs2) = A2 (ts1, bs)
                in
                    (ts2, merge bs1 bs2)
          end;
```

The yielder **given** retrieves and type checks the only datum in the given transients. The function takes a parameter that is a Sort to document the type of the datum in the transients.

```
fun given (S: Sort) : Yielder =
    fn (ts, bs) =>
          case ts of
              [d]   => S d
           |  _     => raise Failure;
```

The yielder GIVEN retrieves and type checks the n^{th} datum in the given transients. Action semantics uses the notation “given d # n” for this operation.

```
fun GIVEN (S:Sort) (n:int) : Yielder =
    fn (ts, bs) =>
          S (take (ts, n));
```

2.4.2 Declarative Facet

The primitive action **bind** :: Token * Yielder → Action produces the binding of its first argument to its second. The first parameter of type Token represents an identifier.

```
fun bind (t:Token) (y:Yielder) : Action =
    fn (ts, bs) =>
          let val dat = Bindable (y (ts, bs))
            in
               (nil, assoc t dat)
          end;
```

Although the signature of bind shows it as uncurried, we choose to define bind as a curried function in ML.

The primitive action **rebind** reproduces the received bindings.

```
val rebind : Action = fn (ts, bs) => (nil, bs);
```

The yielder **boundto** :: Sort * Token → Yielder retrieves and type checks the datum bound to its second argument, an identifier portrayed as a token.

```
infix 2 boundto;

fun (S:Sort) boundto (t:Token) : Yielder =
    fn (ts, bs) =>
        let val d = find bs t
         in
            S d
            handle Failure =>    let val indirection' i = Indirection d
                                  in
                                     case sub (Redirections, i) of
                                               redirection' r    => S r
                                             | notused'          => raise Failure
                                 end
                                 handle Subscript => raise Failure
        end;
```

The datum bound to an identifier may be a bindable or an indirection, which is used to implement a recursive declaration. In the above action, (find bs t) retrieves the datum bound to t in bs. (S d) type checks to see if d is of the desired type. If not, (Indirection d) checks to see if d is an indirection in the case that t represents a procedure or a function identifier. If so, the redirection table is searched to retrieve the datum.

In action semantics Bindable behaves as a subsort of all sorts Sort and boundto is restricted to bindable entities. The implementation of Bindable as a function (not a type) in ML forces us use Sort, a type in ML, to specify the first parameter for boundto. These remarks also apply to the ML function storedin defined later.

The signature for boundto in [Mosses92] has the form

boundto :: Bindable * Yielder → Yielder.

But Token is a subsort of Datum, which is a subsort of Yielder, since a Datum always yields itself when evaluated. So we simplify the implementation by using the sort Token where identifiers are expected.

The declarative action combinator **moreover** allows the performance of the two actions to be interleaved. Both actions use the transients and the bindings passed to the combined action. The transients given by the combined action are the transients given by the first action concatenated with those given by the second. The bindings produced by the combined action are the bindings produced by the first action overlaid by those produced by the second.

```
infix 1 moreover;

fun (A1:Action) moreover (A2:Action) : Action =
    fn (ts, bs) =>
          let val (ts1, bs1) = A1 (ts, bs)
              and (ts2, bs2) = A2 (ts, bs)
               in
                     (ts1 @ ts2, overlay bs2 bs1)
          end;
```

The declarative action combinator **hence** performs the first action using the transients and the bindings passed to the combined action and then performs the second action using the transients given to the combined action and the bindings produced by the first action. The combined action gives the transients given by the first action concatenated with those given by the second. The bindings produced by the combined action are those produced by the second action.

```
infix 1 hence;

fun (A1:Action) hence (A2:Action) : Action =
    fn (ts, bs) =>
          let val (ts1, bs1) = A1 (ts, bs);
               val (ts2, bs2) = A2 (ts, bs1)
            in
               (ts1 @ ts2, bs2)
          end;
```

The declarative action combinator **before'** performs the first action using the transients and the bindings passed to the combined action and then performs the second action using the transients given to the combined action and the bindings received by the combined action overlaid by those produced by the first action. The transients given by the combined action are those given by the first action concatenated with those produced by the second. The combined action produces the bindings produced by the first action overlaid with those produced by the second.

```
infix 1 before';

fun (A1:Action) before' (A2:Action) : Action =
    fn (ts, bs) =>
          let val (ts1, bs1) = A1 (ts, bs);
               val (ts2, bs2) = A2 (ts, overlay bs1 bs)
            in
               (ts1 @ ts2, overlay bs2 bs1)
          end;
```

The action "**furthermore** A" is the same as "rebind moreover A". The action performs A, giving the transients given by A and producing the bindings produced by A overlaying with those received by the action.

```
fun furthermore (A:Action) : Action =
                                        rebind moreover A;
```

2.4.3 Imperative Facet

The primitive action **store** :: Yielder * Yielder → Action stores the storable value yielded by its first argument in the cell yielded by its second argument.

```
fun store (y:Yielder) (c:Yielder) : Action =
    fn (ts, bs) =>
          let val stble = Storable (y (ts, bs))
              and cell' loc = Cell (c (ts, bs))
           in
               update (Storage, loc, stored' stble);
               (nil, empty)
          end
          handle Subscript => raise Failure;
```

The yielder **storedin** :: Sort * Yielder → Yielder retrieves the datum stored in the cell yielded by its second argument, expecting it to be of the type specified by the first argument. The first parameter of storedin is restricted to Storable in action semantics.

```
infix 2 storedin;

fun (S:Sort) storedin (c:Yielder) : Yielder =
    fn (ts, bs) =>
          let val cell' loc = Cell (c (ts, bs))
           in
               case sub (Storage, loc) of
                  stored' stble   => S stble
                | undefined'      => raise Failure
                | unused'         => raise Failure
          end
          handle Subscript => raise Failure;
```

The primitive action **allocateacell** searches the storage to look for an unused cell, reserves it by initializing it to undefined', and gives the cell.

```
fun allocateacell : Action =
    fn (ts, bs) =>
          let fun loop loc =
               (case sub (Storage, loc) of
                     unused'   =>    (update (Storage, loc, undefined');
                                      ([cell' loc], empty)
                    | _ =>       loop (loc + 1))
               handle Subscript => raise Failure
           in
               loop 2
          end;
```

Note that the action starts searching from storage location 2. Storage location 0 is reserved for input, and location 1 is reserved for output. In action semantics, allocateacell is a composite action defined in terms of more primitive functional and imperative actions. We specify it directly to simplify the implementation.

The primitive action **deallocate** :: Yielder → Action marks the cell yielded by its argument as unused' and, therefore makes the cell available for reuse.

```
fun deallocate (c:Yielder) : Action =
    fn (ts, bs) =>
          let val cell' loc = Cell (c (ts, bs))
            in
               update (Storage, loc, unused');
               (nil, empty)
          end
          handle Subscript => raise Failure;
```

2.4.4 Reflective Facet

The primitive action **enact** :: Yielder → Action activates the action encapsulated in the abstraction yielded by its argument.

```
fun enact (a:Yielder) : Action =
    fn (ts, bs) =>
          let val abstraction' (A0, ts0, bs0) = Abstraction (a (ts, bs))
            in
               A0 (ts0, bs0)
          end;
```

The action incorporated in the abstraction is executed with the transients and bindings that are packaged in the abstraction.

The yielder **abstractionof** :: Action → Yielder encapsulates its argument action into an abstraction with empty transients and bindings.

```
fun abstractionof (A:Action) : Yielder =
                    fn (ts, bs) => abstraction' (A, nil, empty);
```

Actually action notation has abstractionof supply no transients or bindings to the abstraction, but if no transients or bindings have been incorporated in the abstraction by enaction-time, the enclosed action is performed using empty transients and/or bindings. We include empty transients and bindings at abstraction-time as an implementation decision.

The yielder **closureof** :: Yielder → Yielder encapsulates current bindings in the abstraction presented to closureof. Performing this operation when constructing the abstraction corresponds to a procedure or function declaration that enforces static scoping by attaching the binding in effect at the time of the declaration to the abstraction. A second application of closureof is not allowed to change the bindings.

```
fun closureof (a:Yielder) : Yielder =
    fn (ts, bs) =>
          case a (ts, bs) of
                abstraction' (A, ts1, empty)    => abstraction' (A, ts1, bs)
              | abstraction' (A, ts1, bs1)      => abstraction' (A, ts1, bs1)
              | _                               => raise Failure;
```

The yielder **applicationof** :: Yielder * Yielder → Yielder attaches the argument list yielded by its second parameter as the transients that will be given to the action encapsulated in the abstraction yielded by its first parameter when that action is enacted. As with closureof, a second utilization of applicationof on an abstraction cannot change the transients.

```
fun applicationof (a:Yielder) (y:Yielder) : Yielder =
    fn (ts, bs) =>
          let val arglist = ArgumentList (y (ts, bs))
           in
              case a (ts, bs) of
                    abstraction' (A, nil, bs1)    => abstraction' (A, [arglist], bs1)
                  | abstraction' (A, ts1, bs1)    => abstraction' (A, ts1, bs1)
                  | _                             => raise Failure
          end;
```

Note that the yielder y should evaluate to an argument list, a tagged list of argument values, and that this list is given as transients to the abstraction yielded by evaluating the yielder a when it is enacted.

2.4.5 Basic Facet

The basic primitive action **complete** simply terminates, passing no transients and no bindings.

```
val complete : Action = fn (ts, bs) => (nil, empty);
```

The basic primitive actions **fail** and **escape** both raise exceptions.

```
val fail : Action = fn (ts, bs) => raise Failure;

exception Escape;

val escape : Action = fn (ts, bs) => raise Escape;
```

The basic action combinator **or** performs either action with the current transients and bindings, and if the chosen action fails, the other alternative is performed with the original transients and bindings.

```
infix 1 or;

fun (A1:Action) or (A2:Action) : Action =
            fn (ts, bs) =>  A1 (ts, bs)
                            handle Failure => A2 (ts, bs);
```

In this implementation, the first alternative action is always chosen first; only if and when it fails will the second alternative be performed.

The basic action combinator **andthen** performs the first action and then performs the second. Both actions use the transients and the bindings passed to the combined action. The transients given by each action are concatenated and given by the combined action. The bindings produced by each action are merged and produced by the combined action.

```
infix 1 andthen;

fun (A1:Action) andthen (A2:Action) : Action =
   fn (ts, bs) =>
        let val (ts1, bs1) = A1 (ts, bs);
            val (ts2, bs2) = A2 (ts, bs)
         in
            (ts1 @ ts2, merge bs1 bs2)
        end;
```

The basic action combinator **and'** allows the performance of the two actions to be interleaved. Both actions use the transients and the bindings passed to the combined action. The transients given by each action are concatenated and given by the combined action. The bindings produced by each action are merged and produced by the combined action.

```
infix 1 and';

fun (A1:Action) and' (A2:Action) : Action = A1 andthen A2;
```

In the ML implementation of action notation, and' is implemented exactly the same as andthen, so that the first action is always performed before the second with no interleaving.

The action **unfolding** :: Action → Action performs its argument action, but whenever the dummy action unfold is encountered, the argument action is performed in place of unfold.

```
fun unfolding (A:Action) : Action =
                furthermore (bind "unfold'" (abstractionof A)) hence A;
```

In the ML implementation, the abstraction of A is bound to the special identifier “unfold'” and this binding overlaying the bindings received by “unfolding A” is passed to A. Therefore, whenever the dummy action unfold is encountered in A, the abstraction of A can be obtained from the bindings and enacted with the transients and bindings current at the time of the unfold.

The primitive action **unfold** is a dummy action, standing for the argument action of the innermost enclosing unfolding.

```
val unfold : Action =
    fn (ts, bs) =>
          case (Abstraction boundto "unfold'") (ts, bs) of
                abstraction' (A, ts1, bs1)     => A (ts, bs)
              | _                              => raise Failure;
```

The actions unfolding and unfold are used to describe indefinite iteration, the **while** command in Δ. Inside a performance of unfolding, an invocation of unfold has the effect of restarting the original action.

2.4.6 Hybrid Action Notation

The action **indirectlybind** :: Token * Yielder → Action takes two parameters, an identifier and a yielder, and produces the binding of the identifier to an indirection, augmenting the redirection table with the indirection initialized to refer to the datum yielded by evaluating the yielder.

```
fun indirectlybind (t:Token) (y:Yielder) : Action =
    fn (ts, bs) =>
        let fun loop n =
             (case sub (Redirections, n) of
                  notused'  => n
                | _         => loop (n + 1))
            handle Subscript => raise Failure
         in
            let val dat = y (ts, bs)
                and i    = loop 0
             in
               case dat of
                    unknown'    =>  (update (Redirections, i, redirection' unknown');
                                     (nil, assoc t (indirection' i)))
                  | _           =>  let val dat' = Bindable (dat)
                                     in
                                        update (Redirections, i, redirection' dat');
                                        (nil, assoc t (indirection' i))
                                    end
            end
        end;
```

The local function loop searches the redirection table for the next available indirection. The action “indirectlybind t y” evaluates y and calls the local function loop to perform the search. The datum yielded by evaluating y may either be unknown' or a bindable; in either case, the redirection table is updated and the binding of the identifier t to the indirection is given with the empty transient.

The action **redirect** takes two arguments, an identifier and a yielder, and changes the indirection bound to the identifier to refer to the datum yielded by evaluating the yielder.

```
fun redirect (t:Token) (y:Yielder) : Action =
    fn (ts, bs) =>
        let val indirection' i = (Indirection boundto t) (ts, bs)
            and dat = y (ts, bs)
         in
           case dat of
             unknown'   =>  (update (Redirections, i, redirection' unknown');
                             (nil, empty))
           | _          =>  let val dat' = Bindable (dat)
                              in
                                update (Redirections, i, redirection' dat');
                                (nil, empty)
                            end
        end;
```

The action **recursivelybind** takes two arguments, an identifier and a yielder, and produces the binding of the identifier to an indirection and at the same time augments the redirection table with the indirection, initialized to refer to the datum yielded by evaluating the yielder using the current bindings overlaid by the indirection binding for the identifier.

```
fun recursivelybind (t:Token) (y:Yielder) : Action  =
                furthermore (indirectlybind t (yield unknown')) hence
                ((redirect t y) and' (bind t (Indirection boundto t)));
```

In the above action, “furthermore (indirectlybind t (yield (unknown')))” produces all the given bindings plus the binding of t to an indirection “indirection' i” that is initialized to unknown'. All these bindings are given to “(redirect t y) and' (bind t (Indirection boundto t))”. The subaction “redirect t y” updates the contents of indirection i, which is bound to t in the given bindings, to contain the value yielded by evaluating y. The purpose of the second subaction “bind t (Indirection boundto t)” is to reproduce the binding of t to the indirection i as the bindings of the action. Recall that the action combinator hence only produces the bindings produced by its second action. The action recursivelybind is necessary for handling self-referential bindings. Examples will be seen later when we specify the action semantics of procedures and functions that may contain recursive calls. The ML code for action primitives, action combinators, and related yielders can be found under “structure Actions” in Appendix C.

2.4.7 Auxiliary Actions

We need some auxiliary yielders and actions that are defined specifically for this implementation of the programming language Δ. They will be described briefly next. A complete implementation of the auxiliary operations in ML can be found under "structure DataEquations2" in Appendix C.

The Primitive Allocator

In the imperative facet, an action allocateacell is defined to allocate a memory cell. An abstraction is needed to encapsulate this action so that the abstraction can be bound to simple type identifiers. When a variable declaration with these types is elaborated, the abstraction can be enacted to allocate a cell for the variable. Therefore, **primitiveallocator** is defined to encapsulate the action of allocateacell.

```
val primitiveallocator : Yielder = closureof (abstractionof (allocateacell));
```

Storage Functions

The action store and the yielder storedin are defined in the imperative facet to store a simple value in a cell and retrieve a simple value stored in a cell, respectively. But Δ also employs compound data types, arrays and records. Therefore, more general forms of store and storedin are needed. The action **assignto** can store any value (simple or compound) in its corresponding variable (cell, array variable, or record variable), and the yielder **valueassignedto** will retrieve the corresponding value stored in any type of Variable. See Appendix C for the ML code of these operations.

```
fun valueassignedto (y:Yielder) : Yielder = ... ;
fun assignto (y1:Yielder, y2:Yielder) : Action = ... ;
```

Input and Output

The following functions, also specified in "structure DataEquations2", are defined to handle Δ input and output.

- The action **allocateinputcell** allocates the cell (cell' 0), which is created to hold the input file, a string of characters.

```
val allocateinputcell : Action =... ;
```

- The action **allocateoutputcell** allocates the cell (cell' 1), which is delegated for output file.

```
val allocateoutputcell : Action = ... ;
```

- The yielder **endofinput** checks whether the input cell is empty.

  val endofinput : Yielder = ... ;

- The yielder **nextcharacter** returns the first character currently stored in the input cell.

  val nextcharacter : Yielder = ... ;

- The action **skipacharacter** removes the first character stored in the input cell.

  val skipacharacter : Action = ... ;

- The action **skipaline** removes all the characters until an end-of-line character is found and then removes the end-of-line character as well.

  val skipaline : Action = ... ;

- The action **skipblanks** removes all blank characters until a non-blank character is found in the input cell.

  val skipblanks : Action = ... ;

- The action **readanunsignedinteger** reads and consumes the characters forming an unsigned integer in the input cell and gives the corresponding integer value.

  val readanunsignedinteger : Action = ... ;

- The action **readasignedinteger** reads and consumes a signed integer in the input cell and gives the integer value.

  val readasignedinteger : Action = ... ;

- The action **rewrite** sets the output cell to empty.

  val rewrite : Action = ... ;

- The action **writechar** appends the character yielded by its argument to the end of output.

  fun writechar (ch:Yielder) : Action = ... ;

- The action **writeunsignedint** writes to output the characters corresponding to the unsigned integer yielded by its argument.

  fun writeunsignedint (i:Yielder) : Action = ... ;

- The action **writesignedint** writes to output the signed integer given by its argument.

```
fun writesignedint (i:Yielder) : Action = ... ;
```

3. Specification of Δ Using Action Semantics

Since actions are implemented as functions that map the information passed to the action into the information passed out of the action, they are executable and can thereby be viewed as providing an interpreter for a programming language defined using the notation of action semantics. The abstract syntax of Δ in ML can be found under “structure TriangleSyntax” in Appendix C, and the complete action semantic definition of Δ in ML can be found in the module “structure TriangleSemantics” in the same appendix.

Program

The output produced by executing a Δ program given particular input, a file represented as an ML string of characters, is taken to be the meaning of that program. Before a Δ program can be executed, the input cell needs to be allocated and initialized to contain the given text (the input to the program). The output cell also needs to be allocated and initialized to empty text, and a standard environment, which provides the initial bindings for all Δ programs, needs to be elaborated. At the end of executing the program, the text stored in the output cell is given as the result of the program. The following action is defined to run a Δ program.

```
fun run (prog' C) =
            (allocateinputcell and' allocateoutputcell)
        andthen
            (store (given Text) inputcell and' rewrite)
        andthen
            (elaborateStandardEnvironment hence execute C)
        andthen
            (give (Text storedin outputcell));
```

3.1 Commands

The execution of a command is modeled as an action that either completes or diverges, although it may update the storage as well; the outcome of the action performance may depend on the current bindings and the current state of storage. Therefore, the semantic function for commands is defined as a function that takes the following form.

```
execute :: Command -> Action   [completing I diverging I storing]
                               [using current bindings I current storage]
```

The preceding signature shows how action notation can restrict the sort of actions produced by the execute function by specifying the possible outcome and income of the actions. The modified sort of actions has the form "Action[outcome][income]". See [Mosses92] for more details on sorts of actions.

Below we give an informal description of each Δ command followed by its semantic equation in action notation in ML. Observe how closely the action notation resembles the informal description. The ML code requires more parentheses than standard action notion, whose rules of operator precedence reduce the number of parentheses. Also, action notation permits multiword identifiers, such as "and then", which cannot be expressed in ML. In spite of these conventions, the ML translation of action notation resembles the original quite closely.

- An empty command

  Do nothing.

  ```
  execute (emptycmd') = complete;
  ```

- An assignment command: V **:=** E

  First the variable name needs to be identified and the expression E evaluated, and then the variable identified by V is updated in storage with the value yielded by E.

  ```
  execute (assign' (V, E)) =
                      (identify V and' evaluate E)
                  then'
                      assignto (GIVEN Value 2, GIVEN Variable 1);
  ```

  Performing "identify V" gives the variable denoted by V and "evaluate E" gives the value yielded by evaluating E. Each of these operations will be described soon.

- An anonymous block with local declarations command: **let** D **in** C

  Execute C in an environment with the bindings produced by elaborating D overlaying the current bindings.

  ```
  execute (letcmd' (D, C)) =
                                  furthermore (elaborate D)
                              hence
                                  execute C;
  ```

  "elaborate D" produces the bindings obtained by elaborating the declarations in D.

- An if command: **if** E **then** C_1 **else** C_2

  The expression E is evaluated first. If it evaluates to true, then C_1 is executed; otherwise C_2 is executed.

  ```
  execute (ifcmd' (E, C1, C2)) =
                  evaluate E
              then'
                      ((check (given TruthValue is TRUE) andthen execute C1)
                  or
                       (check (given TruthValue is FALSE) andthen execute C2));
  ```

  One and only one of the check actions succeeds.

- A while command: **while** E **do** C

  The expression E is evaluated first. If its value is true, C is executed and then the while command is started again; otherwise the while command terminates.

  ```
  execute (while' (E, C)) =
          unfolding
                  (evaluate E
              then'
                      ((check (given TruthValue is TRUE)
                                  andthen execute C andthen unfold)
                  or
                      (check (given TruthValue is FALSE) andthen complete)));
  ```

  The phrase “andthen complete” may be omitted. It simply provides symmetry to the or construct.

- A sequential command: C_1**;** C_2

  Execute C_1 and then execute C_2.

  ```
  execute (seqcmd' (C1, C2)) =
                      execute C1 andthen execute C2;
  ```

- An anonymous block without local declarations command: **begin** C **end**

  Execute C.

  ```
  execute (block' C) =
                          execute C;
  ```

- A procedure call command: t **(** Args **)**

  The actual parameter sequence is evaluated to yield an argument list, and then the procedure is invoked with that argument list.

```
execute (proccall' (t, Args)) =
            givearguments Args
        then'
            enact (applicationof (Procedure boundto t) (given ArgumentList));
```

  The operation "givearguments Args" evaluates the expressions in Args and makes the values yielded into an argument list.

  The Triangle programming language allows parameters of four kinds: value parameters (really constant parameters), variable parameters, procedure parameters, and function parameters. The actual individual parameters are given by an action giveargument defined as follows.

```
fun giveargument (valarg' E) =
                              evaluate E

l  giveargument (vararg' V) =
                              identify V
                          then'
                              give (given Variable)

l  giveargument (procarg' t) =
                              give (Procedure boundto t)

l  giveargument (funarg' t) =
                              give (Function boundto t)
```

3.2 Expressions

The evaluation of an expression is modeled as an action that either gives a value or diverges; the value given by the action performance may depend on the current bindings and the current state of storage. Therefore, the semantic function for expressions is defined as a function of the following type.

```
evaluate :: Expression -> Action    [giving a Value l diverging]
                                    [using current bindings l current storage]
```

The description and the semantic equation in action notation for each form of Δ expressions are given below. Again, note how closely the action notation resembles the informal description.

- Literal: N or C

  If an expression consists of an integer literal N or a character C, evaluating the expression merely gives the integer value N or the character value C, respectively.

```
evaluate (intval' N) =
                    give (yield (integer' N));

evaluate (char' C) =
                    give (yield (char' C));
```

  Since give expects a yielder as its parameter, the data integer' N and char' C must be converted to yielders using yield.

- Variable Name: V

  If the expression consists of a variable name V, evaluating the expression gives the value bound to V (if V is declared as a constant) or the value stored in the variable bound to V (if V is declared as a variable).

```
evaluate (name' V) =
            identify V
        then'
            (give (given Value) or give (valueassignedto (given Variable)));
```

  In the above action, "identify V" gives the object bound to V, either a value or a variable. If a value is given, that value is given by the combined action; otherwise the value assigned to the variable is given by the combined action. Recall that the action combinator or performs either action and if the chosen action fails, the other alternative will be performed.

- Function Call: t (Args)

  The evaluation of a function call is very similar to the execution of a procedure call command. The actual parameter sequence Args is evaluated to yield an argument list, and then the function t is invoked with the argument list.

```
evaluate (funcall' (t, Args)) =
            givearguments Args
        then'
            enact (applicationof (Function boundto t) (given ArgumentList));
```

See the action semantics for a procedure call for a description of the method of giving actual parameters.

- Binary Operation: E_1 O E_2

  The evaluation of a binary operation entails performing the operation of the binary operator O with the values obtained by evaluating E_1 and E_2 as its two arguments.

```
evaluate (binaryop' (O, E1, E2)) =
                 (evaluate E1 and' evaluate E2)
             then'
                 enact (applicationof (Function boundto id(O))
                                              ((argunitlist (GIVEN Value 1)) catto
                                               (argunitlist (GIVEN Value 2))));
```

  In the above action, “id (O)” returns the name of the operator O, and “argunitlist (given Value)” makes the given value into a singleton argument list. The function catto concatenates the two lists. Remember a function is an abstraction. The binding of predefined operations to their meanings takes place in the elaboration of the standard environment, for example:

```
val elaborateStandardEnvironment : Action =
    :
    (bind (id "+") (binaryoperator
        (give (sum ((GIVEN Integer 1), (GIVEN Integer 2)))))) and'
    (bind (id "-") (binaryoperator
        (give (difference ((GIVEN Integer 1), (GIVEN Integer 2)))))) and'
    (bind (id "*") (binaryoperator
        (give (product ((GIVEN Integer 1), (GIVEN Integer 2)))))) and'
    :
```

  where

```
fun binaryoperator (A:Action) : Yielder =
      closureof
          (abstractionof
                   (((give (headof (given ArgumentList)))
               and'
                    (give (headof (tailof (given ArgumentList)))))
              then' A));
```

- Unary Operation: O E

  The evaluation of an unary operation involves performing the operation of the unary operator O with the value yielded by evaluating the expression E as the only argument.

```
evaluate (unaryop' (O, E)) =
                        evaluate E
                    then'
                        enact (applicationof (Function boundto id(O))
                                                (argunitlist (given Value));
```

Unary operations are bound in the standard environment in a manner similar to binary operations. See Appendix C for details.

- Parenthesized Expression: (E)

  A parenthesized expression is evaluated by simply evaluating E.

```
evaluate (paren' E) = evaluate E;
```

- Block Expression: **let** D **in** E

  A block expression is evaluated by evaluating E in an environment with the bindings produced by elaborating D overlaying the current bindings.

```
evaluate (letexp' (D, E)) =
                        furthermore (elaborate D)
                    hence
                        evaluate E;
```

- Conditional Expression: **if** E **then** E_1 **else** E_2

  For an if expression, the expression E is evaluated first. If it evaluates to true, the value of E_1 is returned; otherwise the value of E_2 is returned.

```
evaluate (ifexp' (E, E1, E2)) =
                evaluate E
            then'
                    ((check (given TruthValue is TRUE) andthen evaluate E1)
                or
                     (check (given TruthValue is FALSE) andthen evaluate E2));
```

- Record-Aggregate: RA

  The evaluation of a record-aggregate involves building a record value by evaluating the expression of each constituent field of the record.

```
evaluate (recaggr' RA) = evaluateRecord RA

evaluateRecord (nil) = give (yield (recordvalue' nil))
evaluateRecord ((t, E) :: RA) =
        (evaluate E and' evaluateRecord RA)
    then'
        give (unitrecord (t, GIVEN Value 1) joinedto GIVEN RecordValue 2);
```

The action "evaluateRecord ((t, E) :: RA)" is the action that does most of the work here. A record-aggregate is a list of identifier-expression pairs. This action takes the head of the list, evaluates its expression E, and calls itself to process the tail of the list. It then makes the identifier t and the value given by evaluating E into a unit record, and joins this record to the record value given by evaluating the tail, giving this joined record. The purpose of the action "evaluateRecord (nil)" is to handle the situation when the end of the list is reached. All it does is to give the empty record.

- Array-Aggregate: AA

  The evaluation of an array-aggregate involves building an array value by evaluating the expression corresponding to each element of the array.

```
evaluate (arraggr' AA) = evaluateArray AA;

evaluateArray (nil) = give (yield (arrayvalue' nil));

evaluateArray (E :: AA) =
        (evaluate E and' evaluateArray AA)
    then'
        give (unitarray (GIVEN Value 1) abuttedto GIVEN ArrayValue 2);
```

  Array aggregation is handled in a manner similar to record aggregation. See Appendix C for definitions of the utility operations used to construct records and arrays, namely unitrecord, joinedto, unitarray, and abuttedto.

3.3 Declarations

The elaboration of a declaration is modeled as an action that either produces bindings or diverges (a constant declaration with a function call in its expression may diverge if execution of the function diverges). The bindings produced by the action performance may depend on the current bindings and the current state of storage. Therefore, the semantic function for a declaration is defined as a function of the following type.

```
elaborate :: Declaration -> Action    [binding | diverging]
                                      [using current bindings | current storage]
```

Again we give a description and the semantic equation in action notation for each form of Δ declaration below.

- Constant Declaration: **const** t ~ E

  Elaborating a constant declaration involves binding the identifier t to the value yielded by evaluating E.

```
elaborate (constdec' (t, E)) =
                        evaluate E
                    then'
                        bind t (given Value);
```

- Variable Declaration: **var** t **:** T

  The elaboration of a variable declaration results in binding the identifier t to a Variable (a cell, record variable, or array variable) that can accommodate an entity of type T.

```
elaborate (vardec' (t, T)) =
                        allocatevariable T
                    then'
                        bind t (given Variable);
```

  The operation "allocatevariable T" will allocate enough memory locations and make them into a variable structure that can accommodate an entity of type T.

- Subprogram Declaration: **proc** t **(** Fmls **)** ~ C or **func** t **(** Fmls **) :** T ~ E

  A subprogram declaration is elaborated by binding the identifier t to an abstraction that encapsulates the execution of the procedure body C or the evaluation of the function body E, respectively. The abstraction incorporates an environment containing the bindings of the formal parameters Fmls to the actual parameters provided at procedure or function call time overlaying the current bindings at time of declaration including the binding produced by the subprogram declaration.

```
elaborate (procdec' (t, Fmls, C)) =
    recursivelybind t (closureof
                            (abstractionof
                                    (furthermore (bindparameters Fmls)
                                hence
                                    execute C)));

elaborate (fundec' (t, Fmls, T, E)) =
    recursivelybind t (closureof
                              (abstractionof
                                     (furthermore (bindparameters Fmls)
                                   hence
                                     evaluate E)));
```

Recall that “recursivelybind t y” first binds t to an indirection that is initialized to unknown'. It then evaluates the yielder y. The binding of t to the indirection will be given to the evaluation of y, in this case, “closureof (abstractionof (.....))”. All “closureof (abstractionof (.....))” does is to attached the current bindings to the abstraction. This binding includes the binding of the subprogram name t to the indirection.

Next “recursivelybind t y” updates the indirection to contain the value yielded by y, in this case, the subprogram abstraction. The binding produced by the entire action is the binding of t to the indirection, which now contains the abstraction of the subprogram. Therefore, it is possible for the subprogram to call itself when it is enacted.

Remember, “furthermore A” abbreviates “rebind moreover A”.

Formal parameters are bound to identifiers using the function bindparameters, which calls bindparameter for each of the formal parameters.

```
fun bindparameter (valparam' (t, T)) =
        bind t (given Value)

l  bindparameter (varparam' (t, T)) =
        bind t (given Variable)

l  bindparameter (procparam' (t, FPS)) =
        bind t (given Procedure)

l  bindparameter (funparam' (t, FPS, T)) =
        bind t (given Function)
```

- Type Declaration: **type** t ~ T

The elaboration of a type declaration involves binding the identifier t to an allocator that will allocate an appropriate variable for the type when it is enacted.

```
elaborate (typedec' (t, T)) =
                  bind t (closureof (abstractionof (allocatevariable T)));
```

- Sequential Declaration: D_1 **;** D_2

  A sequential declaration is processed by elaborating D_1 before elaborating D_2.

```
elaborate (seqdec' (D1, D2)) =
                    elaborate D1 before' elaborate D2;
```

  The semantics of the action combinator before' specifies that a declaration may refer to declarations occurring earlier in the block.

4. Running the System

The ML implementation of action semantics has been run and tested using Standard ML of New Jersey, Version 0.75 on a Sun Sparc workstation and Version 0.93 on an IBM RS/6000. Standard ML of New Jersey may be obtained over the Internet using “ftp research.att.com” followed by a directory change “cd dist/ml”.

The semantic prototyping system uses ML-lex and ML-yacc as a scanner and parser. Appendix D contains a listing of the lex and yacc files. The parts of the system are assembled by elaborating a file, called load.sml, that contains definitions of functions that initialize the system and a definition of the function go used to run the interpreter.

```
go : string -> string -> unit
```

For example, the Triangle program “sums” computes and print the sums of the positive integers, the squares of the integers, and the cubes of the integers up to the input value. The program “sums” is listed below followed by a transcript of its execution by the interpreter.

```
let
   proc sumof (func f (n: Integer) : Integer,
                cnt: Integer,
                var sum: Integer) ~
      let
         var i : Integer
       in
         begin
            i := 0;  sum := 0;
            while i < cnt do
               begin
                  i := i + 1;
                  sum := sum + f (i)
```

```
                    end;
                put (' '); putint (sum); put (' ')
            end;

    func g1 (n: Integer) : Integer ~ n;
    func g2 (n: Integer) : Integer ~ n * n;
    func g3 (n: Integer) : Integer ~ n * n * n;

    var count: Integer;
    var s1: Integer;
    var s2: Integer;
    var s3: Integer
  in
    begin
        getint (var count);
        sumof (func g1, count, var s1);
        sumof (func g2, count, var s2);
        sumof (func g3, count, var s3)
    end

     - go "sums" "10";

     OUTPUT:
      55  385  3025

     val it = () : unit
```

The following steps create the semantic prototyping system for Triangle:

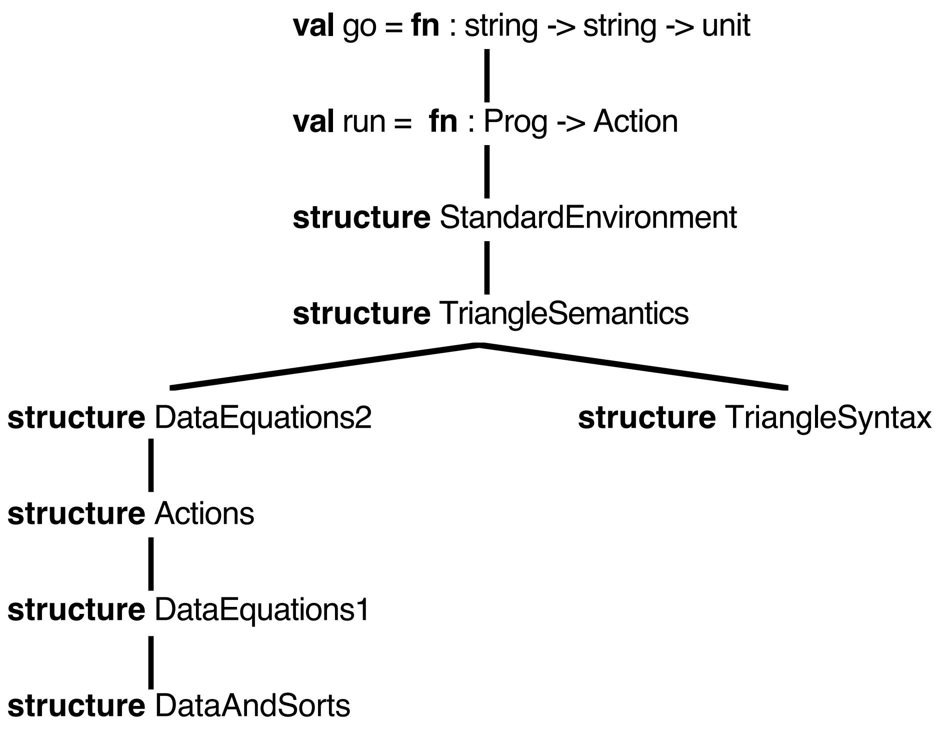
```
cd tools                      -- Move to directory "tools" in SML directory
% sml
- use "lexgen/lexgen.sml";
- use "mlyacc/smlyacc.sml";
- use "mlyacc/base.sml";
- exportML "Bimage";

% Bimage                      -- An sml image with lexgen.sml, smlyacc.sml,
val it = true : bool          -- and base.sml loaded.

- use "load.sml";                -- Generates scanner and parser,
                                 -- loads triangle.sml, triangle.grm.sig,
[opening load.sml]               -- triangle.lex.sml, and triangle.grm.sml,
1 shift/reduce conflict          -- and elaborates several utility
val it = () : unit               -- functions including go (see Appendix D).
Number of states = 40
Number of distinct rows = 12
Approx. memory size of trans. table = 1548 bytes
val it = () : unit
```

```
[opening triangle.sml]
      :                             -- A very long series of responses
      :                             -- as scanner and parser are generated
      :                             -- and structures are elaborated
open StandardEnvironment
val run = fn : Prog -> Action       -- Function for running programs
[closing triangle.sml]
val it = () : unit
[opening triangle.grm.sig]
      :                             -- More responses
[closing triangle.grm.sml]
val it = () : unit
      :                             -- More responses
structure triangleParser : PARSER
val parse = fn : string -> triangleParser.result * …
val convert = fn : string list -> Datum list
val printresult = fn : Datum list -> unit
val cleanStorage = fn : int -> unit
val cleanRedirection = fn : int -> unit
val go = fn : string -> string -> unit   -- Function for controlling interpreter
[closing load.sml]
val it = () : unit
```

The diagram below shows the dependencies among the structures that make up the ML program for the Triangle interpreter.


val go = **fn** : string -> string -> unit
val run = **fn** : Prog -> Action
structure StandardEnvironment
structure TriangleSemantics
structure DataEquations2
structure TriangleSyntax
structure Actions
structure DataEquations1
structure DataAndSorts

5. Conclusion

Generally formal methods of specifying programming language semantics, such as denotational semantics, structural operational semantics, and axiomatic semantics, are difficult to use. See [Slonneger94] for descriptions of these formal methods. The conciseness and notational density of these methods make accurate specifications hard to create, read, and modify. In contrast, action semantics reflects the operational concept of program execution as understood by programmers. Recall how closely the action semantic definition of a command execution, an expression evaluation, or a declaration elaboration resembles its informal description.

Unlike other methods, action semantics uses English-like notation that is easy to read. Although entirely formal, action notation can be read and understood at an informal level without a mastery of a large set of cryptic symbols and notational conventions. Readers can concentrate on the semantics of the programming language from their first exposure to the specification.

Action semantics definitions use a modular style in which semantic equations are reusable in the specifications of any programming languages that share similar language constructs. In contrast to denotational semantics where a small change in the programming language that is being specified can result in major changes in the specification, action semantics definitions scale up with little modification as the programming language is extended.

Once one has a complete implementation of one programming language, as we do here, most of the implementation can be reused when defining another language. New sorts of data can be easily added to the data notation without affecting any of the already existing definitions. The implementation of the action notation provided by action semantics will not change. The only place where some modifications are needed is in the syntax and some of the semantic definitions of the new language.

To test these assertions, we assigned a class project of extending Triangle with a loop…end loop command, an exit command, a repeat command, a return command and a pointer data type with appropriate operations. These additions to Triangle proved fairly straightforward to implement.

All these merits make action semantics easier to understand and more accessible to programmers than other formal methods of specifying the semantics of programming languages. This report illustrates that the readability and understandability of action semantics carries over into a prototype implementation in ML. Transforming a formal specification into a working implementation furnishes us with an excellent tool for testing and experimenting with the formal methods of action semantics. Furthermore, programming action notation can provide insight into the semantics of primitive actions and action combinators that goes deeper than informal descriptions without reaching the notational density and complexity of other formal specifications.

The goals of this report have been to introduce the basics of action semantics and to show the feasibility of translating a formal specification of a programming language in action semantics into a prototype implementation of the language written in ML. The implementation of Δ shows that such semantic prototyping can be smoothly carried out using action semantics as the formal specification method and ML as the implementation language.

Source files for the semantic interpreter may be obtained from the second author by email (slonnegr@cs.uiowa.edu).

References

[Mosses92]
Peter D. Mosses, *Action Semantics*, Cambridge University Press, 1992.

[Slonneger94]
Ken Slonneger and Barry L. Kurtz, *Formal Syntax and Semantics of Programming Languages: A Laboratory-Based Approach*, Addison-Wesley, 1994.

[Watt86]
David A. Watt, “Executable Semantic Descriptions”, *Software - Practice and Experience*, 16.1, 1986.

[Watt91]
David A. Watt, *Programming Language Syntax and Semantics*, Prentice Hall International, 1991.

Appendix A: BNF for Δ

<Program> ::= <Command>

<Command> ::= <single-Command> | <Command> **;** <single-Command>

<single-Command> ::=
| <V-name> **:=** <Expression>
| <Identifier> **(** <Actual-Parameter-Sequence> **)**
| **begin** <Command> **end**
| **let** <Declaration> **in** <single-Command>
| **if** <Expression> **then** <single-Command> **else** <single-Command>
| **while** <Expression> **do** <single-Command>

<Expression> ::= <secondary-Expression>
| **let** <Declaration> **in** <Expression>
| **if** <Expression> **then** <Expression> **else** <Expression>

<secondary-Expression> ::= <primary-Expression>
| <secondary-Expression> <Operator> <primary-Expression>

<primary-Expression> ::= <Integer-Literal>
| <Character-Literal> | <V-name>
| <Identifier> **(** <Actual-Parameter-Sequence> **)**
| <Operator> <primary-Expression>
| **(** <Expression> **)**
| { <Record-Aggregate> }
| **[** <Array-Aggregate> **]**

<Record-Aggregate> ::= <Identifier> **~** <Expression>
| <Identifier> **~** <Expression> **,** <Record-Aggregate>

<Array-Aggregate> ::= <Expression> | <Expression> **,** <Array-Aggregate>

<V-name> ::= <Identifier> | <V-name> **.** <Identifier>
| <V-name> **[** <Expression> **]**

<Declaration> ::= <single-Declaration>
| <Declaration> **;** <single-Declaration>

```
<single-Declaration> ::= const <Identifier> ~ <Expression>
        | var <Identifier> : <Type-denoter>
        | proc <Identifier> ( <Formal-Parameter-Sequence> ) ~
                    <single-Command>
        | func <Identifier> ( <Formal-Parameter-Sequence> )
                    : <Type-denoter> ~ <Expression>
        | type <Identifier> ~ <Type-denoter>

<Formal-Parameter-Sequence> ::=
        | <proper-Formal-Parameter-Sequence>

<proper-Formal-Parameter-Sequence> ::= <Formal-Parameter>
        | <Formal-Parameter> , <proper-Formal-Parameter-Sequence>

<Formal-Parameter> ::= <Identifier> : <Type-denoter>
        | var <Identifier> : <Type-denoter>
        | proc <Identifier> ( <Formal-Parameter-Sequence> )
        | func <Identifier> ( <Formal-Parameter-Sequence> )
                    : <Type-denoter>

<Actual-Parameter-Sequence> ::=
        | <proper-Actual-Parameter-Sequence>

<proper-Actual-Parameter-Sequence> ::= <Actual-Parameter>
        | <Actual-Parameter> , <proper-Actual-Parameter-Sequence>

<Actual-Parameter> ::= <Expression> | var <V-name> | proc <Identifier>
        | func <Identifier>

<Type-denoter> ::= <Identifier>
        | array <Integer-Literal> of <Type-denoter>
        | record <Record-Type-denoter> end

<Record-Type-denoter> ::= <Identifier> ~ <Type-denoter>
        | <Identifier> ~ <Type-denoter> , <Record-Type-denoter>

<Identifier> ::= <Letter> {<Letter-Digit>}

<Operator> ::= <Op-character> {<Op-character>}
```

<Integer-Literal> ::= <Digit> | <Digit> {<Digit>}

<Character-Literal> ::= ' <Graphic> '

<Letter-Digit> ::= <Letter> | <Digit>

<Graphic> ::= <Letter> | <Digit> | <Op-character>
| **space** | **tab** | **.** | **:** | **;** | **,**
| **~** | **(** | **)** | **[** | **]** | { | } | **_** | **|** | **!** | ' | ` | " | **#** | **$**

<Letter> ::= **a** | **b** | **c** | **d** | **e** | **f** | **g** | **h** | **i** | **j** | **k** | **l** | **m**
| **n** | **o** |**p** | **q** | **r** | **s** | **t** | **u** |**v** | **w** |**x** | **y** | **z**
| **A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** | **I** | **J** | **K** | **L** | **M**
| **N** | **O** | **P** | **Q** | **R** | **S** | **T** |**U** |**V** | **W** | **X** | **Y** | **Z**

<Digit> ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

<Op-character> ::= **+** | **-** | ***** | **/** | **=** | **<** | **>** | \ | **&** | @ | **%** | **^** | **?**

Appendix B: Abstract Syntax for Δ

* Programs

```
Prog  ::= Cmd
```

* Commands

```
Cmd  ::= Vname := Expr
       | Identifier ( Arg* )                 -- procedure call
       | begin Cmd end
       | let Dec in Cmd
       | if Expr then Cmd else Cmd
       | while Expr do Cmd
       | Cmd ; Cmd
       |                                     -- empty  command
```

* Expressions

```
Expr  ::= let Dec in Expr
        | if Expr then Expr else Expr
        | Expr Operator Expr                 -- binary operator
        | Integer-Literal
        | Character-Literal
        | Vname
        | Identifier ( Arg * )               -- function call
        | Operator Expr                      -- unary operator
        | ( Expr )
        | { (Identifier ~ Expr) + }          -- record aggregate
        | [ Expr + ]                         -- array aggregate
```

* Names

```
Vname  ::= Identifier
         | Vname . Identifier                -- field in record
         | Vname [ Expr ]              -- element of array
```

* Declarations

```
Dec  ::= Dec ;  Dec
       | const  Identifier ~  Expr
       | var  Identifier :  Typ

       | proc  Identifier (  Param* )  ~  Cmd
       | func  Identifier (  Param* )  :  Typ ~  Expr
       | type  Identifier ~  Typ
```

*** Parameters**

```
Param  ::= Identifier :  Typ
         | var  Identifier :  Typ
         | proc  Identifier (  Param* )
         | func  Identifier (  Param* )  :  Typ

Arg  ::= Expr
       | var  Vname
       | proc  Identifier
       | func  Identifier
```

*** Type-denoters**

```
Typ  ::= Identifier
       | array  Integer-Literal of  Typ
       | record  (Identifier ~ Typ) + end
```

*** Lexicon**

Identifier, Integer-Literal, Character-Literal, and **Operator** are regarded as atomic and not further specified here.

Appendix C: ML Definition of Action Semantics and Δ

```
(*******************************************************************)
(*********  Implementing the Action Semantics for Triangle  *****************)

infix 6 \;

structure DataAndSorts =
struct
        open Array;

exception Failure;

(*****  DATUM  *****)

type Token = string;

datatype Datum =
          truthvalue'             of bool
        | integer'                of int
        | char'           of string
        | arrayvalue'  of Datum list             (*  Value list             *)
        | arrayvariable'          of Datum list                  (*  Variable list                    *)
        | recordvalue'of (Token * Datum) list          (*  (Token * Value) list          *)
        | recordvariable'         of (Token * Datum) list        (*  (Token * Variable) list      *)
        | argumentlist'           of Datum list                  (*  Argument list                    *)
        | text'           of Datum list                  (*  Char list                        *)
        | cell'                   of int
        | indirection'   of int
        | unknown'
(*      | abstraction'  of Action * Transients * Bindings;                         *)
        | abstraction'  of ((Datum list * (Token * Datum) list) ->
                                                    (Datum list * (Token * Datum) list))
                               *  (Datum list)
                               *  ((Token * Datum) list);

        (* Types Action, Transients, and Bindings have not been defined yet. *)

(*****  TRANSIENTS  *****)

type Transients = Datum list;

(*****  BINDINGS  *****)

type Bindings = (Token * Datum) list;
```

```
(*****  ACTIONS  *****)

type Action = Transients * Bindings  -> Transients * Bindings;


(*****  YIELDERS  *****)

type Yielder = Transients * Bindings -> Datum;

(***************   SORTS   ***************)

type Sort = Datum -> Datum;

fun (S1:Sort) \ (S2:Sort) =                         (*  sort join  *)
        fn (d: Datum) => S1 d
                handle Failure => S2 d;

(***  TRUTHVALUES  ***)

val TruthValue : Sort =
         fn (truthvalue' tr) => truthvalue' tr
                |  _           => raise Failure;

(***  INTEGERS  ***)

val Integer : Sort =
        fn (integer' i) => integer' i
                |  _     => raise Failure;

(***  CHARACTERS  ***)

val Char : Sort =
        fn (char' ch) => (case size (ch) of
                                1    => char' ch
                              |  _    => raise Failure)
         |  _   => raise Failure;

(***  CELLS  ***)

val Cell : Sort =
        fn (cell' c)      => cell' c
         |  _             => raise Failure;

(***  INDIRECTIONS  ***)

val Indirection : Sort =
        fn (indirection' i)       => indirection' i
         |  _                     => raise Failure;
```

```
(*** ARRAY-VALUES & ARRAY-VARIABLES ***)

val ArrayValue : Sort =
        fn (arrayvalue' v)     => arrayvalue' v
         | _                   => raise Failure;

val ArrayVariable : Sort =
        fn (arrayvariable' v)     => arrayvariable' v
         | _                      => raise Failure;

(*** RECORD-VALUES & RECORD_VARIABLES ***)

val RecordValue : Sort =
        fn (recordvalue' v     => recordvalue' v
         | _                   => raise Failure;

val RecordVariable : Sort =
        fn (recordvariable' v)    => recordvariable' v
         | _                      => raise Failure;

(************* KINDS OF VALUES *************)

(*** VALUES ***)

val Value : Sort = TruthValue \ Integer \ Char \ ArrayValue \ RecordValue;

(*** VARIABLES ***)

val Variable : Sort = Cell \ ArrayVariable \ RecordVariable;

(*** ABSTRACTIONS ***)

val Abstraction : Sort =
        fn (abstraction' (A, ts, bs))     => abstraction' (A, ts, bs)
         | _                              => raise Failure;

(*** PROCEDURES ***)

val Procedure : Sort = Abstraction;

(*** FUNCTIONS ***)

val Function : Sort = Abstraction;

(*** ALLOCATOR ***)

val Allocator : Sort = Abstraction;

(*** ARGUMENTS ***)

val Argument : Sort = Value \ Variable \ Procedure \ Function;

(*** ARGUMENT LISTS ***)
```

```
val ArgumentList : Sort =
        fn (argumentlist' lis)      => argumentlist' lis
          | _                       => raise Failure;

(***  TEXT  ***)

val Text : Sort =
        fn (text' lis)    => text' lis
          | _             => raise Failure;

(***  BINDABLES  ***)

val Bindable : Sort = Value \ Variable \ Procedure \ Function \ Allocator \ Indirection;

(***  STORABLES  ***)

val Storable : Sort = TruthValue \ Integer \ Char \ Text;

(***************     STORAGE   ***************)

datatype State = unused' | undefined' | stored' of Datum;
val StorageSize = 1000;
val Storage = array (StorageSize, unused');

(***************   REDIRECTIONS   ***************)

datatype RedirectionState = unknown' | redirection' of Datum;
val RedirectionSize = 1000;
val Redirections = array (RedirectionSize, notused');

end;  (*  structure DataAndSorts  *)


   (**********************************************************************)

infix 4 is;
infix 7 modulo;
infix 4 isLessThan;
infix 4 isGreaterThan;
infix 6 abuttedto;
infix 6 joinedto;
infix 6 catto;

structure DataEquations1 =
struct
        open DataAndSorts;
```

```
(*************** Transients = Datum list ***************)

fun take (ts:Transients, n:int) : Datum =              (* the nth datum in   *)
        case ts of                                     (* transient ts       *)
                nil  => raise Failure
              | h::t => if n = 1 then h
                               else if n > 1 then take (t, n-1)
                                      else raise Failure;

(*************** Bindings = (Token * Datum) list ***************)

val empty : Bindings = [];                             (* empty set of bindings    *)

fun assoc (t:Token) (d:Datum) : Bindings =
        [(t d)];                                       (* singleton binding of t to d *)

fun find (bs:Bindings) (t: Token) : Datum =            (* Datum bound to t in      *)
        case bs of                                     (* bindings bs, raising     *)
                nil              => raise Failure      (* Failure if no such  *)
              | (t', dat)::rest => if t' = t then dat    (* binding exists          *)
                                   else find rest t;

exception IdeClash;

fun merge (bs1:Bindings) (bs2:Bindings) : Bindings =   (* Combine bindings     *)
        let fun disjoint (b1, b2) =                    (* bs1 and bs2,         *)
              case b1 of                               (* raising Failure if   *)
                   nil            => true              (* identifiers clash    *)
                 | (t, _)::rest   => let val dat = find b2 t
                                      in
                                        raise IdeClash
                                     end
                                     handle Failure    => disjoint (rest, b2)
                                          | IdeClash   => false
          in
              if disjoint (bs1, bs2)  then (bs1 @ bs2)
                                      else raise Failure
        end;

fun overlay (bs1:Bindings) (bs2:Bindings) : Bindings =
        (bs1 @ bs2);                                   (* Combine bs1 and bs2   *)
                                                       (* with bs1 overriding bs2  *)
```

```
(*************** Yielder = Transients * Bindings -> Datum   ***************)

fun yield (d:Datum) : Yielder = fn (ts, bs) => d;

fun (y1:Yielder) is (y2:Yielder) : Yielder =
      fn (ts, bs) =>
            case (y1 (ts, bs), y2 (ts, bs)) of
                    (truthvalue' tr1, truthvalue' tr2) => truthvalue' (tr1 = tr2)
                  | (integer' n1, integer' n2)         => truthvalue' (n1 = n2)
                  | (char' c1, char' c2)               => truthvalue' (ord (c1) = ord (c2))
                  | (arrayvalue' a1, arrayvalue' a2)   =>
                        let fun eqlis nil nil = truthvalue' true
                              | eqlis (h1::t1) (h2::t2) =
                                    (case ((yield h1) is (yield h2)) (ts, bs) of
                                            truthvalue' true => eqlis t1 t2
                                          | _                => truthvalue' false)
                              | eqlis _ _     = truthvalue' false
                         in
                              eqlis a1 a2
                        end
                  | (recordvalue' r1, recordvalue' r2) =>
                        let fun eqlis nil nil = truthvalue' true
                              | eqlis ((I1, dat1)::t1) ((I2, dat2)::t2) =
                                    (if I1 = I2 then
                                                (case ((yield dat1) is (yield dat2)) (ts, bs) of
                                                        truthvalue' true => eqlis t1 t2
                                                      | _                => truthvalue' false)
                                               else truthvalue' false)
                              | eqlis _ _     = truthvalue' false)
                          in
                              eqlis r1 r2
                        end
                  | (cell' loc1, cell' loc2)           => truthvalue' (loc1 = loc2)
                  | (_, _)                             => raise Failure;

(***  TRUTHVALUES  ***)

val FALSE : Yielder =
        fn (ts, bs) => truthvalue' false;
```

```
val TRUE : Yielder =
        fn (ts, bs) => truthvalue' true;

fun not' (y:Yielder) : Yielder =
        fn (ts, bs) =>
                let val truthvalue' tr = TruthValue (y (ts, bs))
                  in
                        truthvalue' (not tr)
                end;

fun both (y1:Yielder, y2:Yielder) : Yielder =
        fn (ts, bs) =>
                let val truthvalue' tr1 = TruthValue (y1 (ts, bs))
                   and truthvalue' tr2 = TruthValue (y2 (ts, bs))
                 in
                        truthvalue' (tr1 andalso tr2)
                end;

fun either (y1:Yielder, y2:Yielder) : Yielder =
        fn (ts, bs) =>
                let val truthvalue' tr1 = TruthValue (y1 (ts, bs))
                   and truthvalue' tr2 = TruthValue (y2 (ts, bs))
                 in
                        truthvalue' (tr1 orelse tr2)
                end;

(***  INTEGERS  ***)

fun successor (y:Yielder) : Yielder =
        fn (ts, bs) =>
                let val integer' i = Integer (y (ts, bs))
                 in
                        integer' (i + 1)
                end;

fun predecessor (y:Yielder) : Yielder =
        fn (ts, bs) =>
                let val integer' i = Integer (y (ts, bs))
                 in
                        integer' (i - 1)
                end;

fun negation (y:Yielder) : Yielder =
        fn (ts, bs) =>
                let val integer' i = Integer (y (ts, bs))
                 in
                        integer' (~i)
                end;

fun sum (y1:Yielder, y2:Yielder) : Yielder =
        fn (ts, bs) =>
                let val integer' n1 = Integer (y1 (ts, bs))
                   and integer' n2 = Integer (y2 (ts, bs))
                  in
                        integer' (n1 + n2)
                end;

fun difference (y1:Yielder, y2:Yielder) : Yielder =
        fn (ts, bs) =>
                let val integer' n1 = Integer (y1 (ts, bs))
                   and integer' n2 = Integer (y2 (ts, bs))
                 in
                        integer' (n1 - n2)
                end;

fun product (y1:Yielder, y2:Yielder) : Yielder =
        fn (ts, bs) =>
                let val integer' n1 = Integer (y1 (ts, bs))
                  and integer' n2 = Integer (y2 (ts, bs))
                 in
                        integer' (n1 * n2)
                end;

fun quotient (y1:Yielder, y2:Yielder) : Yielder =
        fn (ts, bs) =>
                let val integer' n1 = Integer (y1 (ts, bs))
                   and integer' n2 = Integer (y2 (ts, bs))
                 in
                        integer' (n1 div n2)
                end;

fun (y1:Yielder) modulo (y2:Yielder) : Yielder =
        fn (ts, bs) =>
                let val integer' n1 = Integer (y1 (ts, bs))
                   and integer' n2 = Integer (y2 (ts, bs))
                  in
                        integer' (n1 mod n2)
                end;

fun (y1:Yielder) isLessThan (y2:Yielder) : Yielder =
        fn (ts, bs) =>
                let val integer' n1 = Integer (y1 (ts, bs))
                   and integer' n2 = Integer (y2 (ts, bs))
                 in
                        truthvalue' (n1 < n2)
                end;

fun (y1:Yielder) isGreaterThan (y2:Yielder) : Yielder =
        fn (ts, bs) =>
                let val integer' n1 = Integer (y1 (ts, bs))
                   and integer' n2 = Integer (y2 (ts, bs))
                 in
                        truthvalue' (n1 > n2)
                end;
```

```
(***  CHARACTERS  ***)

val eolnchar : Yielder = fn (ts, bs) => char' "\n";

fun decodeof (y:Yielder) : Yielder =
        fn (ts, bs) =>
                let val integer' i = Integer (y (ts, bs))
                  in
                        char' (chr (i))
                end
                handle Chr => raise Failure;

fun codeof (y:Yielder) : Yielder =
        fn (ts, bs) =>
                let val char' ch = Char (y (ts, bs))
                 in
                        integer' (ord (ch))
                end
                handle Ord => raise Failure;

fun blank (y:Yielder) : Yielder =
        fn (ts, bs) =>
                case y (ts, bs) of
                          char' " "  => truthvalue' true
                        | char' "\t" => truthvalue' true
                        | char' ch  => truthvalue' false
                        | _            => raise Failure;

fun decimal (y:Yielder) : Yielder =
        fn (ts, bs) =>
                case y (ts, bs) of
                          char' "0"    => truthvalue' true
                        | char' "1"    => truthvalue' true
                        | char' "2"    => truthvalue' true
                        | char' "3"    => truthvalue' true
                        | char' "4"    => truthvalue' true
                        | char' "5"    => truthvalue' true
                        | char' "6"    => truthvalue' true
                        | char' "7"    => truthvalue' true
                        | char' "8"    => truthvalue' true
                        | char' "9"    => truthvalue' true
                        | char' ch     => truthvalue' false
                        | _            => raise Failure;

fun decimaldigit (y:Yielder) : Yielder =
        fn (ts, bs) =>
                case y (ts, bs) of
                          integer' 0     => char' "0"
                        | integer' 1     => char' "1"
                        | integer' 2     => char' "2"
                        | integer' 3     => char' "3"
                        | integer' 4     => char' "4"
                        | integer' 5     => char' "5"
```

```
                              l  integer' 6      => char' "6"
                              l  integer' 7      => char' "7"
                              l  integer' 8      => char' "8"
                              l  integer' 9      => char' "9"
                              l  _       => raise Failure;

fun decimalvalueof (y:Yielder) : Yielder =
          fn (ts, bs) =>
                    case y (ts, bs) of
                                char' "0"     => integer' 0
                              l  char' "1"     => integer' 1
                              l  char' "2"     => integer' 2
                              l  char' "3"     => integer' 3
                              l  char' "4"     => integer' 4
                              l  char' "5"     => integer' 5
                              l  char' "6"     => integer' 6
                              l  char' "7"     => integer' 7
                              l  char' "8"     => integer' 8
                              l  char' "9"     => integer' 9
                              l  _             => raise Failure;

(***  CELLS  ***)

val inputcell : Yielder = fn (ts, bs) => cell' 0;

val outputcell : Yielder = fn (ts, bs) => cell' 1;

(***  ARRAY-VALUES  &  ARRAY-VARIABLES  ***)

fun unitarray (y:Yielder) : Yielder =
          fn (ts, bs) =>
                    let val v = Value (y (ts, bs))
                               in  arrayvalue' [v]
                    end
                    handle Failure =>       let val var = Variable (y (ts, bs))
                                              in  arrayvariable' [var]
                                            end;

fun (y1:Yielder) abuttedto (y2:Yielder) : Yielder =
          fn (ts, bs) =>
          case (y1 (ts, bs), y2 (ts, bs)) of
                      (arrayvalue' aval1, arrayvalue' aval2)
                                            => arrayvalue' (aval1 @ aval2)
                    l  (arrayvariable' avar1, arrayvariable' avar2)
                                            => arrayvariable' (avar1 @ avar2)
                    l  (_, _)               => raise Failure;

fun componentof (y1:Yielder, y2:Yielder) : Yielder =
          fn (ts, bs) =>
                    let fun get (i:int) (lst:Datum list) =
                              case lst of
                                          nil   => raise Failure
                                        l  h::t  => if i > 0 then get (i - 1) t
                                                    else if i = 0 then h
```

```
                                                              else raise Failure
                  in
                        case (y1 (ts, bs), y2 (ts, bs)) of
                                  (integer' i, arrayvalue' aval)          => get i aval
                                | (integer' i, arrayvariable' avar)       => get i avar
                                | (_, _)                                  => raise Failure
                end;

fun sizeof (y:Yielder) : Yielder =
        fn (ts, bs) =>
        let fun length nil  = 0
            | length (h::t) = 1 + length t
         in
                case (y (ts, bs)) of
                          arrayvalue' aval        => integer' (length aval)
                        | arrayvariable' avar     => integer' (length avar)
                        | _                       => raise Failure
        end;

(***  RECORD-VALUES  &  RECORD_VARIABLES  ***)

fun unitrecord (t:Token, y:Yielder) : Yielder =
        fn (ts, bs) =>
                let val v = Value (y (ts, bs))
                 in
                        recordvalue' [(t, v)]
                end
                handle Failure =>       let val var = Variable (y (ts, bs))
                                         in
                                                recordvariable' [(t, var)]
                                        end;

fun (y1:Yielder) joinedto (y2:Yielder) : Yielder =
        fn (ts, bs) =>
                case (y1 (ts, bs), y2 (ts, bs)) of
                          (recordvalue' rval1, recordvalue' rval2)
                                                        => recordvalue' (rval1 @ rval2)
                        | (recordvariable' rvar1, recordvariable' rvar2)
                                                        => recordvariable' (rvar1 @ rvar2)
                        | (_, _)                        => raise Failure;

fun fieldof (t:Token, y:Yielder) : Yielder =
        fn (ts, bs) =>
        let fun get (Id:Token) (b:(Token * Datum) list) =
                (case b of
                          nil               => raise Failure
                        | (t, dat)::rest    => (if t = Id then dat
                                                       else get Id rest))
         in
                case (y (ts, bs)) of
                          recordvalue' rval        => get t rval
                        | recordvariable' rvar     => get t rvar
                        | _                        => raise Failure
        end;
```

```
(*** ARGUMENT LISTS & TEXT ***)

val argemptylist : Yielder = fn (ts, bs) => argumentlist' nil;

val textemptylist : Yielder = fn (ts, bs) => text' nil;

fun argunitlist (y:Yielder) : Yielder =
        fn (ts, bs) =>
                let val dat = Argument (y (ts, bs))
                 in
                        argumentlist' [dat]
                end;

fun textunitlist (y:Yielder) : Yielder =
        fn (ts, bs) =>
                let val dat = Char (y (ts, bs))
                 in
                        text' [dat]
                end;

fun  (y1:Yielder) catto(y2:Yielder) : Yielder =
        fn (ts, bs) =>
                case (y1 (ts, bs), y2 (ts, bs)) of
                     (argumentlist' l1, argumentlist' l2)      => argumentlist' (l1 @ l2)
                   | (text' t1, text' t2)                      => text' (t1 @ t2)
                   | (_, _)                                    => raise Failure;

fun isempty (y:Yielder) : Yielder =
        fn (ts, bs) =>
                case y (ts, bs) of
                     argumentlist' nil       => truthvalue' true
                   | argumentlist' (h::t)    => truthvalue' false
                   | text' nil               => truthvalue' true
                   | text' (ch::st)          => truthvalue' false
                   | _                       => raise Failure;

fun headof (y:Yielder) : Yielder =
        fn (ts, bs) =>
                case y (ts, bs) of
                     argumentlist' (h::t)    => h
                   | text' (ch::st)          => ch
                   | _                       => raise Failure;

fun tailof (y:Yielder) : Yielder =
        fn (ts, bs) =>
                case y (ts, bs) of
                     argumentlist' (h::t)    => argumentlist' t
                   | text' (ch::st)          => text' st
                   | _                       => raise Failure;

end; (* structure DataEquations1 *)

(*******************************************************************)
```

```
(************************************************************************)

infix 1 then';
infix 2 boundto;
infix 1 moreover;
infix 1 hence;
infix 1 before';
infix 2 storedin;
infix 1 or
infix 1 andthen;
infix 1 and';
infix 1 trap;

structure Actions =
struct
        open DataEquations1;


(************* The Functional Facet *************)

fun check (t:Yielder) : Action =
        fn (ts, bs) =>
                case t (ts, bs) of
                           truthvalue' true       => (nil, empty)
                        l  truthvalue' false      => raise Failure
                        l  _                      => raise Failure;

val regive : Action =
        fn (ts, bs) => (ts, empty);

fun give (y: Yielder) : Action =
        fn (ts, bs) =>
                let val dat = y (ts, bs)
                  in
                        ([dat], empty)
                end;

fun (A1:Action) then' (A2:Action) : Action =                (*  A1 then A2        *)
        fn (ts, bs) =>
                let val (ts1, bs1) = A1 (ts, bs);
                    val (ts2, bs2) = A2 (ts1, bs)
                 in
                        (ts2, merge bs1 bs2)
                end;
```

```
fun given (S:Sort) : Yielder =                          (*  the given S         *)
        fn (ts, bs) =>
                case ts of
                          [d]    => S d
                        | _      => raise Failure;

fun GIVEN (S:Sort) (n:int) : Yielder =                  (*  the given S#n       *)
        fn (ts, bs) =>
                    S (take (ts, n));

(************* The Declarative Facet  *************)

fun bind (t:Token) (y:Yielder) : Action =               (*  bind t to y *)
        fn (ts, bs) =>
                let val dat = Bindable (y (ts, bs))
                 in
                        (nil, assoc t  dat)
                end;

val rebind : Action =
        fn (ts, bs) => (nil, bs);

fun (S:Sort) boundto ( t:Token) : Yielder =             (*  the S bound to t  *)
        fn (ts, bs) =>
        let val d = find bs t
         in
                S  d
                handle Failure => let val indirection' i = Indirection d
                                   in
                                        case sub (Redirections, i) of
                                              redirection' r     => S r
                                            | notused'           => raise Failure
                                  end
                                  handle Subscript => raise Failure
        end;

fun (A1: Action) moreover (A2:Action) : Action =  (*  A1 moreover A2 *)
        fn (ts, bs) =>
                let val (ts1, bs1) = A1 (ts, bs)
                   and (ts2, bs2) = A2 (ts, bs)
                 in
                        (ts1 @ ts2, overlay bs2 bs1)
                end;
```

```
fun (A1: Action) hence (A2:Action) : Action =                    (*  A1 hence A2      *)
        fn (ts, bs) =>
                let val (ts1, bs1) = A1 (ts, bs);
                    val (ts2, bs2) = A2 (ts, bs1)
                 in
                        (ts1 @ ts2, bs2)
                end;

fun (A1:Action) before' (A2:Action) : Action =                  (*  A1 before A2      *)
        fn (ts, bs) =>
                let val (ts1, bs1) = A1 (ts, bs);
                    val (ts2, bs2) = A2 (ts, overlay bs1 bs)
                 in
                        (ts1 @ ts2, overlay bs2 b1)
                end;

fun furthermore (A:Action) : Action =                            (*  furthermore A     *)
        rebind moreover A;

(************* The Imperative Facet *************)

fun store (y:Yielder) (c:Yielder) : Action =            (*  store y in c         *)
        fn (ts, bs) =>
                let val stble = Storable (y (ts, bs))
                and cell' loc = Cell (c (ts, bs))
                 in
                        update (Storage, loc, stored' stble);
                        (nil, empty)
                end
                handle Subscript => raise Failure;

fun (S:Sort) storedin (c:Yielder) : Yielder =                    (*  the S stored in c   *)
        fn (ts, bs) =>
                let val cell' loc = Cell (c (ts, bs))
                 in
                        case sub (Storage, loc) of
                                  stored' stble   => S (stble)
                                | undefined'      => raise Failure
                                | unused'         => raise Failure
                end
                handle Subscript => raise Failure;

fun deallocate (c:Yielder) : Action =                            (*  deallocate c        *)
        fn (ts, bs) =>
                let val cell' loc = Cell (c (ts, bs))
                 in
                        update (Storage, loc, unused');
                        (nil, empty)
                end;

val allocateacell : Action =                             (*  allocate a cell      *)
        fn (ts, bs) =>
                let fun loop loc =
                    (case sub (Storage, loc) of
                           unused' => (update (Storage, loc, undefined');
                                       ([cell' loc], empty))
                         I _        => loop (loc + 1))
                    handle Subscript => raise Failure
                in
                    loop 2
                end;

(************* Reflective Facet *************)

fun enact (a: Yielder) : Action =                        (*  enact a            *)
        fn (ts, bs) =>
                let val abstraction' (A0, ts0, bs0) = Abstraction (a (ts, bs))
                 in
                    A0 (ts0, bs0)
                end;

fun abstractionof (A:Action) : Yielder =                 (*  abstraction of A   *)
        fn (ts, bs) =>
                abstraction' (A, nil, empty)

fun closureof (a: Yielder) : Yielder =                   (*  closure of a       *)
        fn (ts, bs) =>
                case a (ts, bs) of
                       abstraction' (A, ts1, empty) => abstraction' (A, ts1, bs)
                     I abstraction' (A, ts1, bs1)   => abstraction' (A, ts1, bs1)
                     I _                            => raise Failure;


fun applicationof (a:Yielder) (y:Yielder) : Yielder =    (*  application of a to y  *)
        fn (ts, bs) =>
                let val arglist = ArgumentList (y (ts, bs))
                 in
                   case a (ts, bs) of
                       abstraction' (A, nil, bs1)  => abstraction' (A, [arglist], bs1)
                     I abstraction' (A, ts1, bs1) => abstraction' (A, ts1, bs1)
                     I _                          => raise Failure
                end;
```

```
(************* The Basic Facet *************)

val complete : Action = fn (ts, bs) => (nil, empty);

val fail : Action = fn (ts, bs) => raise Failure;

exception Escape;

val escape : Action = fn (ts, bs) => raise Escape;

fun (A1:Action) or (A2:Action) : Action =           (*  A1 or A2          *)
        fn (ts, bs) => A1 (ts, bs)
                        handle Failure => A2 (ts, bs);

fun (A1:Action) andthen (A2:Action) : Action =      (*  A1 and then A2    *)
        fn (ts, bs) =>
                let val (ts1, bs1) = A1 (ts, bs);
                    val (ts2, bs2) = A2 (ts, bs)
                 in
                        (ts1 @ ts2, merge bs1 bs2)
                end;

fun (A1:Action) and' (A2:Action) : Action =                 (*  A1 and A2          *)
                 A1 andthen A2;

fun (A1:Action) trap (A2:Action) : Action =         (*  A1 trap A2        *)
        fn (ts, bs) => A1 (ts, bs)
                        handle Escape => A2 (ts, bs);

fun unfolding (A: Action) : Action =                (*  unfolding A       *)
        furthermore (bind "unfold'" (abstractionof A)) hence A;

val unfold : Action =                               (*  unfold            *)
        fn (ts, bs) =>
                case (Abstraction boundto "unfold'") (ts, bs) of
                         abstraction' (A, _, _) => A (ts, bs)
                        | _                     => raise Failure;
```

```
(************* Hybrid Actions *************)

fun indirectlybind (t:Token) (y:Yielder) : Action =   (*  indirectly bind t to y  *)
        fn (ts, bs) =>
               let fun loop n =
                     (case sub (Redirections, n) of
                              notused' => n
                           | _           => loop (n + 1))
                     handle Subscript => raise Failure
                in
                     let val dat = y (ts, bs)
                           and i = loop 0
                       in case dat of
                              unknown' =>(update (Redirections, i, redirection' unknown');
                                                (nil, assoc t (indirection' i)))
                           | _              => let val dat' = Bindable dat
                                                in
                                                      update (Redirections, i, redirection' dat');
                                                      (nil, assoc t (indirection' i))
                                              end
                     end
               end;

fun redirect (t:Token) (y:Yielder) : Action =                       (*  redirect t to y        *)
         fn (ts, bs) =>
                   let val indirection' i = (Indirection boundto t) (ts, bs)
                                  and dat = y (ts, bs)
                     in case dat of
                             unknown' => (update (Redirections, i, redirection' unknown');
                                                (nil, empty))
                           | _              => let val dat' = Bindable dat
                                                in
                                                      update (Redirections, i, redirection' dat');
                                                      (nil, empty)
                                             end
                   end;

fun recursivelybind (t:Token) (y:Yielder) : Action =                  (*  recursively           *)
            furthermore (indirectlybind t (yield unknown'))     (*  bind t to y            *)
         hence
            (redirect t y and' bind t (Indirection boundto t));

end; (* structure Actions *)

(**********************************************************************)
```

```
(*********************************************************************)

structure DataEquations2 =
struct
        open Actions;

(***  ALLOCATOR  ***)

val primitiveallocator : Yielder =
        closureof (abstractionof (allocateacell));

(*************   STORAGE   *************)

fun valueassignedto (y:Yielder) : Yielder =         (*  the value assigned to y  *)
      fn (ts, bs) =>
            case y (ts, bs) of
              arrayvariable' av  => let fun get nil aval    = aval
                                      | get (h::t) aval =
                                            let val hval =
                                                  valueassignedto (yield (h)) (ts, bs)
                                              in
                                                  get t (aval @ [hval])
                                            end
                                    in
                                      arrayvalue' (get av nil)
                                  end
      | recordvariable' rv => let fun get nil rval        = rval
                                      |  get ((t, v)::rest) rval =
                                            let val hval =
                                                  valueassignedto (yield (v)) (ts, bs)
                                              in
                                                  get rest (rval @ [(t, hval)])
                                             end
                                      in
                                            recordvalue' (get rv nil)
                                  end
      | cell' c                => (Storable storedin yield (cell' c)) (ts, bs)
      | _                      => raise Failure;
```

```
fun assignto (y1:Yielder, y2:Yielder) : Action =            (*  assign y1 to y2   *)
        fn (ts, bs) =>
                case (y1 (ts, bs), y2 (ts, bs)) of
                          (arrayvalue' aval, arrayvariable' avar)  =>
                                let fun put nil nil          = (nil, empty)
                                    | put (h1::t1) (h2::t2) =
                                                (assignto (yield (h1), yield (h2)) (ts, bs);
                                                put t1 t2)
                                    | put _ _                = raise Failure
                                 in
                                        put aval avar
                                end
                        | (recordvalue' rval, recordvariable' rvar) =>
                                let fun put nil nil = (nil, empty)
                                    | put ((I1, val')::t1) ((I2, var)::t2) =
                                        (if I1 = I2 then
                                                (assignto (yield(val'), yield (var)) (ts, bs);
                                                put t1 t2)
                                          else raise Failure)
                                    | put _ _        = raise Failure
                                 in
                                        put rval rvar
                                end
                        | (_, _)  => store y1 y2 (ts, bs);

(*************   INPUT-OUTPUT   *************)

val allocateinputcell : Action =
        fn (ts, bs) =>
                let val cell' inloc = inputcell (ts, bs)
                 in
                        update (Storage, inloc, undefined');
                        (nil, empty)
                end;

val allocateoutputcell : Action =
        fn (ts, bs) =>
                let val cell' outloc = outputcell (ts, bs)
                 in
                        update (Storage, outloc, undefined');
                        (nil, empty)
                end;

val endofinput : Yielder =
        isempty (Text storedin inputcell);

val nextcharacter : Yielder =
        headof (Text storedin inputcell);

val skipacharacter : Action =
        store (tailof (Text storedin inputcell)) inputcell;
```

```
val skipaline : Action =
        unfolding ((check (endofinput) andthen complete)
                or
                     (check (nextcharacter is eolnchar)) andthen skipacharacter)
                or
                     (check (not' (nextcharacter is eolnchar))) andthen
                                                skipacharacter andthen unfold));

val skipblanks : Action =
        unfolding ((check endofinput) andthen complete)
                or
                    (check (blank nextcharacter) andthen
                                                skipacharacter andthen unfold)
                or
                    (check (not' (blank nextcharacter)) andthen complete));

val readanunsignedinteger : Action =
            (give (decimalvalueof nextcharacter) andthen skipacharacter)
        then'
            unfolding
                        ((check (decimal nextcharacter) andthen
                            ((give (sum (decimalvalueof (nextcharacter),
                                    product (yield (integer' 10), given Integer)))
                                andthen skipacharacter)
                            then' unfold) )
                    or
                        (check (not' (decimal nextcharacter)) andthen
                                                    give (given Integer))
                    or
                        (check endofinput andthen give (given Integer)));

val readasignedinteger : Action =
            (check (nextcharacter is yield (char' "-")) andthen
                skipacharacter andthen
                (readanunsignedinteger then' give (negation (given Integer))))
        or
            (check (not' (nextcharacter is yield (char' "-"))) andthen
                readanunsignedinteger);

val rewrite : Action =
        store textemptylist outputcell;
```

```
fun writechar (ch:Yielder) : Action =
        store ((Text storedin outputcell) catto textunitlist ch) outputcell;

fun writeunsignedint (i:Yielder) : Action =
    fn (ts,bs) =>
            ((check (i isLessThan yield (integer' 10)) andthen
              writechar (decimaldigit i))
        or
             (check (not' (i isLessThan yield (integer' 10))) andthen
              writeunsignedint (quotient (i, yield (integer' 10))) andthen
              writechar (decimaldigit (i modulo yield (integer' 10))))) (ts,bs);

fun writesignedint (i:Yielder) : Action =
            (check (i isLessThan yield (integer' 0))) andthen
                writechar (yield (char' "-")) andthen
                writeunsignedint (negation i))
        or
            (check (not' (i isLessThan yield (integer' 0)))) andthen
                writeunsignedint i);

end; (* structure DataEquations2 *)

(***********************************************************************)
```

```
(*******************************************************************)

structure TriangleSyntax =
struct
        open DataAndSorts;
datatype
        Prog = prog' of Cmd
and     Cmd  = assign'          of Vname * Expr
                | proccall'     of Token * Arg list
                | block'        of Cmd
                | letcmd'       of Dec * Cmd
                | ifcmd'        of Expr * Cmd * Cmd
                | while'        of Expr * Cmd
                | seqcmd'       of Cmd * Cmd
                | emptycmd'
and     Expr =  letexp'         of Dec * Expr
                | ifexp'        of Expr * Expr * Expr
                | binaryop'     of Token * Expr * Expr
                | intval'       of int
                | charval'      of string
                | name'         of Vname
                | funcall'      of Token * Arg list
                | unaryop'      of Token * Expr
                | paren'        of Expr
                | recaggr'      of (Token * Expr) list
                | arraggr'      of Expr list
and     Vname = id'   of Token
                | recid'        of Vname * Token
                | arrid' of Vname * Expr
and     Dec =   seqdec'         of Dec * Dec
                | constdec'     of Token * Expr
                | vardec'       of Token * Typ
                | procdec'      of Token * Param list * Cmd
                | fundec'       of Token * Param list * Typ * Expr
                | typedec'      of Token * Typ
and     Param = valparam'       of Token * Typ
                | varparam'     of Token * Typ
                | procparam' of Token * Param list
                | funparam' of Token * Param list * Typ
and     Arg = valarg'  of Expr
                | vararg'       of Vname
                | procarg'      of Token
                | funarg'       of Token
and     Typ  = type'            of Token
                | arrtype'      of int * Typ
                | rectype'      of (Token * Typ) list

end; (* structure TriangleSyntax *)
(*******************************************************************)
```

```
(*********************************************************************)

structure TriangleSemantics =
struct
        open DataEquations2;
        open TriangleSyntax;

fun
          execute (emptycmd') =
                complete

        l  execute (assign' (V, E)) =
                    (identify V and' evaluate E)
                then'
                    assignto (GIVEN Value 2, GIVEN Variable 1)

        l  execute (proccall' (t, Args)) =
                    givearguments Args
                then'
                    enact (applicationof (Procedure boundto t) (given ArgumentList))

        l  execute (seqcmd' (C1, C2)) =
                    execute C1
                andthen
                    execute C2

        l  execute (block' C) =
                execute C

        l  execute (letcmd' (D, C)) =
                    furthermore (elaborate D)
                hence
                    execute C

        l  execute (ifcmd' (E, C1, C2)) =
                    evaluate E
                then'
                        ((check (given TruthValue is TRUE) andthen execute C1)
                    or
                         (check (given TruthValue is FALSE) andthen execute C2))

        l  execute (while' (E, C)) =
                unfolding
                        (evaluate E
                    then'
                            ((check (given TruthValue is TRUE) andthen
                                execute C andthen unfold)
                        or
                             (check (given TruthValue is FALSE) andthen complete)))
and
          evaluate (intval' N) =
                give (yield (integer' N))

        l  evaluate (charval' C) =
                give (yield (char' C))

        l  evaluate (name' V) =
                    identify V
                then'
                    (give given Value) or give (valueassignedto (given Variable)))

        l  evaluate (funcall' (t, Args)) =
                    givearguments Args
                then'
                    enact (applicationof (Function boundto t) (given ArgumentList))

        l  evaluate (unaryop' (O, E)) =
                    evaluate E
                then'
                    enact (applicationof (Function boundto id(O)) (argunitlist (given Value)))

        l  evaluate (binaryop' (O, E1, E2)) =
                    (evaluate E1 and' evaluate E2)
                then'
                    enact (applicationof (Function boundto id(O))
                        ((argunitlist (GIVEN Value 1)) catto (argunitlist (GIVEN Value 2))))

        l  evaluate (paren' E) =
                evaluate E

        l  evaluate (letexp' (D, E)) =
                    furthermore (elaborate D)
                hence
                    evaluate E

        l  evaluate (ifexp' (E, E1, E2)) =
                    evaluate E
                then'
                        ((check (given TruthValue is TRUE) andthen evaluate E1)
                    or
                         (check (given TruthValue is FALSE) andthen evaluate E2))

        l  evaluate (recaggr' RA) =
                evaluateRecord RA

        l  evaluate (arraggr' AA) =
                evaluateArray AA
and
           evaluateRecord (nil) =
                give (yield (recordvalue' nil))

        l  evaluateRecord ((t, E)::RA) =
                    (evaluate E and' evaluateRecord RA)
                then'
                    give (unitrecord (t, GIVEN Value 1) joinedto GIVEN RecordValue 2)
and
           evaluateArray (nil) =
                give (yield (arrayvalue' nil))
        | evaluateArray (E::AA) =
                  (evaluate E and' evaluateArray AA)
              then'
                  give (unitarray (GIVEN Value 1) abuttedto GIVEN ArrayValue 2)
and
          identify (id' t) =
                  give (Value boundto t)
              or
                  give (Variable boundto t)

        | identify (recid' (V, t)) =
                  identify V
              then'
                      (give (fieldof (t, given RecordValue))
                  or
                       give (fieldof (t, given RecordVariable)))

        | identify (arrid' (V, E)) =
                  (identify V and' evaluate E)
              then'
                      (give (componentof (GIVEN Integer 2, GIVEN ArrayValue 1))
                  or
                       give (componentof (GIVEN Integer 2, GIVEN ArrayVariable 1)))
and
          elaborate (constdec' (t, E)) =
                  evaluate E
              then'
                  bind t (given Value)

        | elaborate (vardec' (t, T)) =
                  allocatevariable T
              then'
                  bind t (given Variable)


        | elaborate (procdec' (t, Fmls, C)) =
              recursivelybind t (closureof
                      (abstractionof
                              (furthermore (bindparameters Fmls)
                          hence
                              execute C)))

        | elaborate (fundec' (t, Fmls, T, E)) =
              recursivelybind t (closureof
                      (abstractionof
                              (furthermore (bindparameters Fmls)
                          hence
                              evaluate E)))

        | elaborate (typedec' (t, T)) =
              bind t (closureof (abstractionof (allocatevariable T)))

        | elaborate (seqdec' (D1, D2)) =
                elaborate D1
            before'
                elaborate D2
and
         bindparameters (nil) =
            complete

       | bindparameters (h::t) =
                (give (headof (given ArgumentList)) then' bindparameter h)
            and'
                (give (tailof (given ArgumentList)) then' bindparameters t)
and
         bindparameter (valparam' (t, T)) =
            bind t (given Value)

       | bindparameter (varparam' (t, T)) =
            bind t (given Variable)

       | bindparameter (procparam' (t, FPS)) =
            bind t (given Procedure)

       | bindparameter (funparam' (t, FPS, T)) =
            bind t (given Function)
and
         givearguments (nil) =
            give argemptylist

       | givearguments (h::t) =
                (giveargument h and' givearguments t)
            then'
                (give ((argunitlist (GIVEN Argument 1)) catto (GIVEN ArgumentList 2)))
and
         giveargument (valarg' E) =
            evaluate E

       | giveargument (vararg' V) =
                identify V
            then'
                give (given Variable)

       | giveargument (procarg' t) =
            give (Procedure boundto t)

       | giveargument (funarg' t) =
            give (Function boundto t)
and
         allocatevariable (type' t) =
            enact (Allocator boundto t)

       | allocatevariable (arrtype' (n, T)) =                (*  Assumes n>0  *)
            (allocatevariable T then' give (unitarray (given Variable)))
          then'
            unfolding
            ((check (sizeof (given ArrayVariable) isLessThan  yield (integer' n))
                    andthen
                ((give (given ArrayVariable) and' allocatevariable T)
                    then'
                (give (GIVEN ArrayVariable 1 abuttedto
                                        unitarray (GIVEN Variable 2)))
                    then' unfold ))
            or
            (check (sizeof (given ArrayVariable) is yield (integer' n))
                andthen give (given ArrayVariable)))

    | allocatevariable (rectype' RT) =
            allocaterecordvariable RT
and
      allocaterecordvariable (nil) =
            give (yield (recordvariable' nil))

    | allocaterecordvariable ((t, T)::rest) =
                (allocatevariable T and' allocaterecordvariable rest)
            then'
                give (unitrecord (t, GIVEN Variable 1) joinedto
                                                GIVEN RecordVariable 2)
and
      id ("\\")    = "not#" (* Indentifiers (Tokens) for operation symbols *)
    | id ("/\\")   = "and#"
    | id ("\\/")   = "or#"
    | id ("+")     = "sum#"
    | id ("-")     = "difference#"
    | id ("*")     = "product#"
    | id ("/")     = "div#"
    | id ("//")    = "mod#"
    | id ("<")     = "less#"
    | id ("<=")    = "lesseq#"
    | id (">")     = "greater#"
    | id (">=")    = "greatereq#"
    | id ("=")     = "eq#"
    | id ("\\=")   = "noteq#"

end; (* structure TriangleSemantics *)

(*********************************************************************)
```

```
(**********************************************************************)

structure StandardEnvironment =
struct
        open TriangleSemantics;

fun unaryoperator (A:Action) : Yielder =
        closureof ( abstractionof
                (give (headof (given ArgumentList)) then' A));

fun binaryoperator (A:Action) : Yielder =
        closureof ( abstractionof
                ((give (headof (given ArgumentList)) and'
                  give (headof (tailof (given ArgumentList)))) then' A));

val eoffunction : Yielder =
        closureof (abstractionof (give (endofinput)));

val eolfunction : Yielder =
        closureof ( abstractionof
                ((check endofinput andthen (give TRUE) or
                 (check (not' endofinput) andthen
                 give (nextcharacter is eolnchar))));

val getprocedure : Yielder =
        closureof ( abstractionof
                ((give (headof (given ArgumentList)) and'
                  (give nextcharacter andthen skipacharacter )) then'
                  store (GIVEN Char 2) (GIVEN Cell 1)));

val putprocedure : Yielder =
        closureof ( abstractionof
                (give (headof (given ArgumentList)) then'
                 writechar (given Char)));

val getintegerprocedure : Yielder =
        closureof ( abstractionof
                ((give (headof (given ArgumentList)) and'
                  skipblanks andthen readasignedinteger) then'
                  store (GIVEN Integer 2) (GIVEN Cell 1)));

val putintegerprocedure : Yielder =
        closureof ( abstractionof
                (give (headof (given ArgumentList)) then'
                 writesignedint (given Integer)));

val geteolprocedure : Yielder =
        closureof ( abstractionof ( skipaline ) );


val puteolprocedure : Yielder =
        closureof ( abstractionof ( writechar eolnchar ) );

val elaborateStandardEnvironment : Action =
        (bind "Boolean" primitiveallocator) and'
        (bind "false" FALSE) and'
        (bind "true" TRUE) and'
        (bind (id "\\") (unaryoperator (give (not' (given TruthValue)))) ) and'
        (bind (id "/\\") (binaryoperator
                (give (both ((GIVEN TruthValue 1), (GIVEN TruthValue 2)))))) and'
        (bind (id "\\/") (binaryoperator
                (give (either ((GIVEN TruthValue 1), (GIVEN TruthValue 2)))))) and'
        (bind "Integer" primitiveallocator) and'
        (bind (id "+") (binaryoperator
                (give (sum ((GIVEN Integer 1), (GIVEN Integer 2)))))) and'
        (bind (id "-") (binaryoperator
                (give (difference ((GIVEN Integer 1), (GIVEN Integer 2)))))) and'
        (bind (id "*") (binaryoperator
                (give (product ((GIVEN Integer 1), (GIVEN Integer 2)))))) and'
        (bind (id "/") (binaryoperator
                (give (quotient ((GIVEN Integer 1), (GIVEN Integer 2)))))) and'
        (bind (id "//") (binaryoperator
                (give ((GIVEN Integer 1) modulo (GIVEN Integer 2))))) and'
        (bind (id "<") (binaryoperator
                 (give ((GIVEN Integer 1) isLessThan (GIVEN Integer 2))))) and'
        (bind (id "<=") (binaryoperator
                (give (not' ((GIVEN Integer 1) isGreaterThan (GIVEN Integer 2)))))) and'
        (bind (id ">") (binaryoperator
                (give ((GIVEN Integer 1) isGreaterThan (GIVEN Integer 2))))) and'
        (bind (id ">=") (binaryoperator
                (give (not' ((GIVEN Integer 1) isLessThan (GIVEN Integer 2)))))) and'
        (bind "Char" primitiveallocator) and'
        (bind "chr" (unaryoperator (give (decodeof (given Integer))))) and'
        (bind "ord" (unaryoperator (give (codeof (given Char))))) and'
        (bind "eof" eoffunction) and'
        (bind "eol" eolfunction) and'
        (bind "get" getprocedure) and'
        (bind "put" putprocedure) and'
        (bind "getint" getintegerprocedure) and'
        (bind "putint" putintegerprocedure) and'  (**)
        (bind "geteol" geteolprocedure) and'
        (bind "puteol" puteolprocedure) and'
        (bind (id "=") (binaryoperator
                (give ((GIVEN Integer 1) is (GIVEN Integer 2))))) and'
        (bind (id "\\=") (binaryoperator
                (give (not' ((GIVEN Integer 1) is (GIVEN Integer 2))))));

end; (* structure StandardEnvironment *)
(*********************************************************************)
```

```
(**********************************************************************)

open StandardEnvironment;

fun run (prog' C) =
            ( allocateinputcell and' allocateoutputcell )
        andthen
            ( store (given Text) inputcell and' rewrite )
        andthen
            ( elaborateStandardEnvironment hence execute C )
        andthen
            ( give (Text storedin outputcell) );

(**********************************************************************)
```

Appendix D: Utility Files for ML Implementation

```
(**********************************************************************)
(*** load.sml    -- elaborates files and functions needed to            ***)
(***                -- define parser and action semantics interpreter   ***)

ParseGen.parseGen "triangle.grm";          (* Run yacc on grammar specification        *)
                                           (*   creating "triangle.grm.sig"            *)
                                           (*   and "triangle.grm.sml"                 *)
LexGen.lexGen "triangle.lex";              (* Run lex on scanner specification *)
                                           (*   creating "triangle.lex.sml"            *)
use "triangle.sml";
use "triangle.grm.sig";
use "triangle.lex.sml";
use "triangle.grm.sml";

structure triangleLrVals =
                        triangleLrValsFun(structure Token = LrParser.Token);
structure triangleLex =
                        triangleLexFun(structure Tokens = triangleLrVals.Tokens);
structure triangleParser =
                        Join(  structure Lex= triangleLex
                                 structure LrParser = LrParser
                                 structure ParserData = triangleLrVals.ParserData);

val parse = fn filename =>
        let   val instrm = open_in filename
              val lexer = triangleParser.makeLexer(fn i => input(instrm,i))
              val _ = triangleLex.UserDeclarations.pos := 1
              val error = fn (e,i:int,_) => output(std_out,filename ^ "," ^
                              " line " ^ (makestring i) ^ ", Error: " ^ e ^ "\n")
          in triangleParser.parse(30,lexer,error,()) before close_in instrm
        end;

fun convert (strlist:string list) =
        case strlist of
           nil      => nil
         | h::t     => (char' h) :: (convert t);

fun printresult (datumlist:Datum list) =
        case datumlist of
           nil                  => output (std_out, "\n\n")
         | (char' ch) :: t      => (output (std_out, ch);
                                     printresult (t));
```

```
fun cleanStorage n =
        let val i = ref 0
          in
                while !i < n do
                        (update (Storage, !i, unused'); i := !i + 1)
        end;

fun cleanRedirection n =
        let val i = ref 0
          in
                while !i < n do
                        (update (Redirections, !i, notused'); i := !i + 1)
        end;

fun go filename instring =
        let val (abs_syn, _) = parse filename
          in (  cleanStorage StorageSize; cleanRedirection RedirectionSize;
                run abs_syn ([text' (convert (explode instring))], empty);
                output (std_out, "\nOUTPUT:\n");
                let val stored' (text' ls) = sub (Storage, 1)
                  in printresult (ls)
                end)
        end;

(***   end   load.sml   *****************************************************)
(**********************************************************************)
```

```
(***********************************************************************)
(******* triangle.lex -- lexer specification for the language Triangle ***********)

type pos = int
type svalue = Tokens.svalue
type ('a, 'b) token = ('a, 'b) Tokens.token
type lexresult = (svalue, pos) token

val pos = ref 1
val error = fn x => output(std_out, x ^ "\n")
val eof = fn () => Tokens.yEOF(!pos, !pos)

%%

%header (functor triangleLexFun(structure Tokens : triangle_TOKENS));

alpha=[A-Za-z];
digit=[0-9];
alphanumeric=[A-Za-z0-9];
graphic=[A-Za-z0-9\ \t+*/=<>&@%^?.:;,~(){}_|!'`"#$-];
ws=[\ \t];
%%

\n                 => (pos := (!pos) + 1; lex());
{ws}+              => (lex());
'{graphic}'        => (Tokens.yQUOTED(explode yytext, !pos, !pos));
"+"                => (Tokens.yOP(yytext, !pos, !pos));
"*"                => (Tokens.yOP(yytext, !pos, !pos));
"-"                => (Tokens.yOP(yytext, !pos, !pos));
"/"                => (Tokens.yOP(yytext, !pos, !pos));
"//"               => (Tokens.yOP(yytext, !pos, !pos));
"\\"               => (Tokens.yOP(yytext, !pos, !pos));
"/\\"              => (Tokens.yOP(yytext, !pos, !pos));
"\\/"              => (Tokens.yOP(yytext, !pos, !pos));
{digit}+           => (Tokens.yNUM(revfold (fn (a,r) => ord(a) - ord("0") + 10*r)
                                  (explode yytext) 0, !pos, !pos));
{alpha}{alphanumeric}*            => (case yytext of
                   "begin"        => Tokens.yBEGIN(!pos, !pos)
                 | "end"          => Tokens.yEND(!pos, !pos)
                 | "if"           => Tokens.yIF(!pos, !pos)
                 | "then"         => Tokens.yTHEN(!pos, !pos)
                 | "else"         => Tokens.yELSE(!pos, !pos)
                 | "while"        => Tokens.yWHILE(!pos, !pos)
                 | "do"           => Tokens.yDO(!pos, !pos)
                 | "let"          => Tokens.yLET(!pos, !pos)
                 | "in"           => Tokens.yIN(!pos, !pos)
                 | "const"        => Tokens.yCONST(!pos, !pos)
                 | "var"          => Tokens.yVAR(!pos, !pos)
                 | "proc"         => Tokens.yPROC(!pos, !pos)
                 | "func"         => Tokens.yFUNC(!pos, !pos)
                 | "type"         => Tokens.yTYPE(!pos, !pos)
                 | "array"        => Tokens.yARRAY(!pos, !pos)
```

```
              | "of"            => Tokens.yOF(!pos, !pos)
              | "record"        => Tokens.yRECORD(!pos, !pos)
              | _               => Tokens.yID(yytext, !pos, !pos));
":="                            => (Tokens.yASSIGN(!pos, !pos));
";"                             => (Tokens.ySEMI(!pos, !pos));
":"                             => (Tokens.yCOLON(!pos, !pos));
","                             => (Tokens.yCOMMA(!pos, !pos));
"."                             => (Tokens.yDOT(!pos, !pos));
"~"                             => (Tokens.yTILDE(!pos, !pos));
"("                             => (Tokens.yLPAREN(!pos, !pos));
")"                             => (Tokens.yRPAREN(!pos, !pos));
"["                             => (Tokens.yLSQUAR(!pos, !pos));
"]"                             => (Tokens.yRSQUAR(!pos, !pos));
"{"                             => (Tokens.yLCURLY(!pos, !pos));
"}"                             => (Tokens.yRCURLY(!pos, !pos));
"="                             => (Tokens.yOP(yytext, !pos, !pos));
"<="                            => (Tokens.yOP(yytext, !pos, !pos));
"<"                             => (Tokens.yOP(yytext, !pos, !pos));
">"                             => (Tokens.yOP(yytext, !pos, !pos));
">="                            => (Tokens.yOP(yytext, !pos, !pos));
"\\="                           => (Tokens.yOP(yytext, !pos, !pos));
.                               => (error ("error: bad identifier "^yytext); lex());

(***  end  triangle.lex  ****************************************************)
(**************************************************************************)
```

```
(**************************************************************************)
(******** triangle.grm -- parser specification for the language Triangle *********)

exception rmQuotesFail;
fun rmquotes (ls) =
  case ls of
    ["'", ch, "'"]      => ch
   | _                  => raise rmQuotesFail

%%
%name triangle       (* "triangle" becomes a prefix in functions *)
%verbose
%eop yEOF ySEMI
%pos int

%term yID of string     | yNUM of int      | yQUOTED of string list
      | yBEGIN          | yEND             | yIF          | yTHEN        | yELSE
      | yWHILE          | yDO              | yLET         | yIN          | yCONST
      | yVAR            | yFUNC            | yPROC        | yTYPE        | yARRAY
      | yOF             | yRECORD          | yASSIGN      | ySEMI        | yCOLON
      | yCOMMA          | yDOT             | yEOF         | yLPAREN      | yRPAREN
      | yLSQUAR         | yRSQUAR          | yLCURLY      | yRCURLY
      | yTILDE          | yOP of string

%nonterm yProgram of Prog
        | yCmds of Cmd              | yCmd of Cmd
        | yExp of Expr              | ySecondary of Expr        | yPrimary of Expr
        | yRecAggr of (string * Expr) list
        | yArrAggr of Expr list
        | yVName of Vname
        | yDecs of Dec              | yDec of Dec
        | yParams of Param list     | yParam of Param
        | yArguments of Arg list    | yArgument of Arg
        | yType of Typ
        | yRecType of (string * Typ) list

%%

yProgram : yCmds                              (prog'(yCmds))

yCmds : yCmd                                  (yCmd)
      | yCmd ySEMI yCmds                      (seqcmd'(yCmd, yCmds))

yCmd  : yVName yASSIGN yExp                   (assign'(yVName, yExp))
      | yID yLPAREN yArguments yRPAREN        (proccall'(yID, yArguments))
      | yID yLPAREN yRPAREN                   (proccall'(yID, nil))
      | yBEGIN yCmds yEND                     (block'(yCmds))
      | yLET yDecs yIN yCmd                   (letcmd'(yDecs, yCmd))
      | yIF yExp yTHEN yCmd yELSE yCmd        (ifcmd'(yExp, yCmd1, yCmd2))
      | yWHILE yExp yDO yCmd                  (while'(yExp,yCmd))
      |                                       (emptycmd')
```

```
yExp : ySecondary                          (ySecondary)
      l yLET yDecs yIN yExp                (letexp'(yDecs, yExp))
      l yIF yExp yTHEN yExp yELSE yExp     (ifexp'(yExp1, yExp2, yExp3))

ySecondary : yPrimary                      (yPrimary)
      l ySecondary yOP yPrimary            (binaryop'(yOP, ySecondary, yPrimary))

yPrimary : yNUM                            (intval'(yNUM))
      l yQUOTED                            (charval'(rmquotes(yQUOTED)))
      l yVName                             (name' yVName)
      l yID yLPAREN yArguments yRPAREN     (funcall'(yID, yArguments))
      l yID yLPAREN yRPAREN                (funcall'(yID, nil))
      l yOP yPrimary                       (unaryop'(yOP, yPrimary))
      l yLPAREN yExp yRPAREN               (paren'(yExp))
      l yLCURLY yRecAggr yRCURLY           (recaggr'(yRecAggr))
      l yLSQUAR yArrAggr yRSQUAR           (arraggr'(yArrAggr))

yRecAggr : yID yTILDE yExp                 ([(yID, yExp)])
      l yID yTILDE yExp yCOMMA yRecAggr    ((yID, yExp)::yRecAggr)

yArrAggr : yExp                            ([yExp])
      l yExp yCOMMA yArrAggr               (yExp::yArrAggr)


yVName : yID                               (id'(yID))
      l yVName yDOT yID                    (recid'(yVName, yID))
      l yVName yLSQUAR yExp yRSQUAR        (arrid'(yVName, yExp))

yDecs : yDec                               (yDec)
      l yDec ySEMI yDecs                   (seqdec'(yDec, yDecs))

yDec : yCONST yID yTILDE yExp              (constdec'(yID, yExp))
      l yVAR yID yCOLON yType              (vardec'(yID, yType))
      l yPROC yID yLPAREN yParams yRPAREN yTILDE yCmd
                                           (procdec'(yID, yParams, yCmd))
      l yPROC yID yLPAREN yRPAREN yTILDE yCmd
                                           (procdec'(yID, nil, yCmd))
      l yFUNC yID yLPAREN yParams yRPAREN yCOLON yType yTILDE yExp
                                           (fundec'(yID, yParams, yType, yExp))
      l yFUNC yID yLPAREN yRPAREN yCOLON yType yTILDE yExp
                                           (fundec'(yID, nil, yType, yExp))
      l yTYPE yID yTILDE yType             (typedec'(yID, yType))

yParams : yParam                           ([yParam])
      l yParam yCOMMA yParams              ([yParam] @ yParams)

yParam : yID yCOLON yType                  (valparam'(yID, yType))
      l yVAR yID yCOLON yType              (varparam'(yID, yType))
      l yPROC yID yLPAREN yParams yRPAREN
                                           (procparam'(yID, yParams))
      l yPROC yID yLPAREN yRPAREN          (procparam'(yID, nil))
      l yFUNC yID yLPAREN yParams yRPAREN yCOLON yType
                                           (funparam'(yID, yParams, yType))
```

```
     | yFUNC yID yLPAREN yRPAREN yCOLON yType
                                              (funparam'(yID, nil, yType))

yArguments : yArgument                        ([yArgument])
     | yArgument yCOMMA yArguments            ([yArgument] @ yArguments)

yArgument : yExp                              (valarg'(yExp))
     | yVAR yVName                            (vararg'(yVName))
     | yPROC yID                              (procarg'(yID))
     | yFUNC yID                              (funarg'(yID))

yType : yID                                   (type'(yID))
     | yARRAY yNUM yOF yType                  (arrtype'(yNUM, yType))
     | yRECORD yRecType yEND                  (rectype'(yRecType))

yRecType : yID yCOLON yType                   ([(yID, yType)])
     | yID yCOLON yType yCOMMA yRecType
                                              ([(yID, yType)] @ yRecType)

(*** end triangle.grm ****************************************************)
(************************************************************************)
```