CSC 530 Lecture Notes Week 5

More on Formal Semantics with Attribute Grammars

I. Attribute semantics of real programming languages

A. Last week's pretty trivial

B. These notes investigate SIL
-- a simple imperative language.

II. Attribute semantics meta-languages

A. Knuth not 100% rigorous.

1. Meta-language not fully formalized.

2. Meta-language conventions must be defined.

Meta-languages, cont’d

B. Syntactic meta-language

1. Based on YACC.

2. $n notation used.

3. Semantic equations use YACC format.

Meta-languages, cont'd

C. Semantic meta-language

1. Based on ML.

2. Attributes are ML types.

3. Semantic equations are ML exprs.

4. Aux functions are ML.

Meta-languages, cont'd

D. Additional notation

1. Basic equation format:

   \$n.attr = expr

   expr is ML with *attribute access terms* of the form

   \$n.attr

2. ML types extended with `nil_T` and `error_T` for all `T`.

III. Circularity in attribute definitions

A. Can arise in practice.

B. E.g.,

```
A : B
    {$1.x = $$.x
     $$.x = $1.x}
```

C. In standard def, circularities render entire def ill-formed.

Circularity, cont'd

D. Eliminated by *attribute splitting*

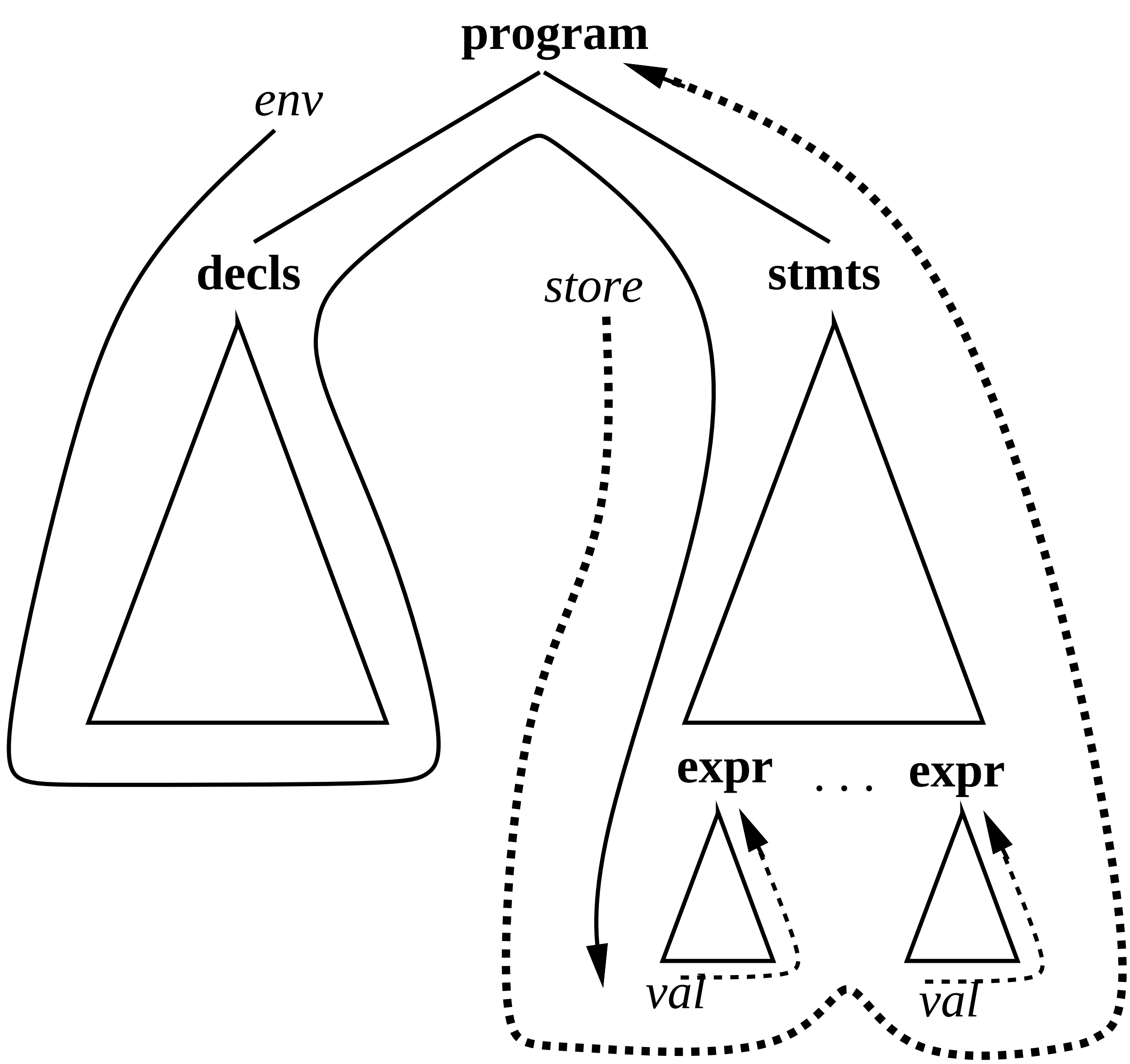
1. Attr *x* decomposed into *x* (inherited) and *x'* (synthesized).

2. Above circular def rewritten:

```
A : B
    {$1.x = $$.x
     $$.x' = $1.x'}
```

E. Attr splitting used in SIL.

IV. Attribute flow in real PLs



Attribute flow, cont'd

A. See Figure 1.

B. Attr eval in one depth-first pass.

C. This is the case with SIL.

D. Certain lang features require > one pass

E. General multi-pass eval discussed in Bochman.

V. Attribute semantics of SIL

```
/*
 * Like Lisp with setq.
 * Diffs:
 *
 *
 * (1) Pascal-like syntax
 *
 *
 * (2) explicit type decls
 *
 *
 * (3) distinguishes between
 *     stmts and exprs
 *
 *
```

Semantics of SIL, cont'd

```
*
* Semantic attributes:
*
* NAME    DESCRIPTION
* ==========================
*
* state   Tuple (env, store)
*
* env     List [ env_binding ...]
*
* store   List [ act_rec ... ]
*
* env_binding
*         Tuple (name, def)
*
* def     One of var_def or
*         fun_def.
*
```

Semantics of SIL, cont’d

```
 *
 * var_def
 *          A type.
 *
 * fun_def
 *         (type, formals, body).
 *
 * formals
 *          [ env_binding ... ]
 *
 * type     One of "integer",
 *          "real", "string",
 *          "boolean", or "OK".
 *
 * body     fn:(env*store)->store’
 *
```

Semantics of SIL, cont'd

```
 *
 * act_rec
 *           [ value_binding, ... ]
 *
 * value_binding
 *           (name, value)
 *
 * value     One of integer or
 *           real or string or
 *           boolean primitives
 *
 * op_fun    fn:(value*value)->value
 *
 * name      A string.
 *
 * nil_X, error_X
 *           Built-in to meta-
 *           language for each
 *           attribute type X
```

Semantics of SIL, cont'd

```
*
* Aux functions:
*
*
* fun assoc(name, alist) =
*     if null(alist) then
*         nil_binding
*     else if name =
*         #1(hd(alist)) then
*             hd(alist)
*     else assoc(name, tl(alist))
*
*
* fun last(l) = hd(nthtail(
*         l, length(l)-1))
*
```

Semantics of SIL, cont'd

```
*
*
* fun butlast(l) =
*     if (null(l) orelse
*         null(tl(l))) then nil
*     else hd(l) :: butlast(tl(l))
*
*
* fun reassign_local(name, value,
*         alist) =
*     if name = #1(hd(alist)) then
*         (name, value) :: tl(alist)
*     else hd(alist) :: reassign(
*         name, value, tl(alist)
*
```

Semantics of SIL, cont'd

```
*
* fun assign(name, value, alist) =
*         (name, value) :: alist
*
*
* fun chk_apply(fun_name,
*         actual_types, env)
*   let
*     val fun_binding =
*         assoc(fun_name, env)
*     val formals =
*         #2(fun_binding)
*     val fun-type =
*         #1(fun_binding)
*   in
*
```

Semantics of SIL, cont'd

```
 *
 *      if chk_bindings(formals,
 *              actuals) then
 *          if fun_type =
 *                  nil_type then
 *              "OK"
 *          else
 *              fun_type
 *      else
 *          error_type
 *    end
 *
```

Semantics of SIL, cont'd

```
*
* fun chk_bindings(formals,
*         actuals) =
*     if formals = nil
*     then true
*     else (hd(formals) =
*         hd(actuals)) and
*         chk_bindings(
*             tl(formals),
*             tl(actuals))
*
```

Semantics of SIL, cont'd

```
*
*
* fun apply(fun_name, actuals,
*           env, store) =
*   let
*     val fun_binding =
*         assoc(fun_name, env)
*     val fun_body =
*         #3(fun_binding)
*     val formals =
*         #2(fun_binding)
*     val bindings =
*         bind(formals, actuals)
*   in
*     fun_body(env, bindings @
*                      store)
*   end
*
```

Semantics of SIL, cont’d

```
*
* fun bind(formals, actuals) =
*     if formals = nil then nil
*     else (hd(formals),
*        hd(actuals)) ::
*          bind(
*            tl(formals),
*            tl(actuals))
*
```

Semantics of SIL, cont'd

```
 *
 * fun functionize(tree,ins,outs) =
 *    a meta-function that trans-
 *    forms an attributed parse
 *    tree denoted by T into a
 *    function
 *         fT(ia<1>*...*ia<m>)->
 *             (sa<1>*...*sa<n>)
 *
 *
 * fun init_env() = []
 */
```

SIL Rules

```
program :
  PROGRAM decls stmts END
    {$2.env = init_env()
     $3.env = $2.env’
     $3.store = nil_store
     $$.state = if $3.type = "OK" then
                    ($2.env’, $3.store’)
                else
                    error_state}
    ;
```

SIL Rules, cont’d

```
decls :
  /* empty */
    {$$.env’ = []}
  | decl ’;’ decls
    {$1.env = $$.env
     $3.env = $1.env’
     $$.env’ = $1.env’ @ $3.env’}
  ;
```

SIL Rules, cont’d

```
decl :
  vardecl
    {$$.env’ = $1.env’}
  | procdecl
    {$1.env = $$.env
     $$.env’ = $1.env’}
  ;
```

SIL Rules, cont’d

```
vardecl :
  VAR vars ’:’ type
    {$2.type = $4.type
     $$.env’ = $2.env’}
  ;
```

SIL Rules, cont'd

```
type :
  INTEGER
    {$$.type = "integer"}
  | REAL
    {$$.type = "real"}
  | CHAR
    {$$.type = "char"}
  | BOOLEAN
    {$$.type = "boolean"}
  ;
```

SIL Rules, cont’d

```
vars :
  var
    {$$.env’ = [($1.name, $$.type)]}
  | var ’,’ vars
    {$$.env’ = $1.env’ @ $3.env’}
  ;
```

SIL Rules, cont'd

```
var :
  IDENTIFIER
    {$$.name = $1.name}
     /* NOTE: The lexer provides
        ident string names. */
    ;
```

SIL Rules, cont’d

```
procdecl :
  PROCEDURE prochdr procbody
    {$3.env = $2.formals @ $$.env
     $$.env’ =
       [($2.name, nil_type,
         $2.formals, $3.fun_body)]}

  | PROCEDURE prochdr ’:’ type procbody
    {$5.env = $$.env
     $$.env’ =
       [($2.name, $4.type,
         $2.formals @
          [($2.name, $4.type)],
       /*  ^^^^^^^^^^^^^^^^^^^ return val
         $5.fun_body)]}
    ;
```

SIL Rules, cont’d

```
prochdr :
  IDENTIFIER ’(’ formals ’)’
    {$$.name = $1.name
     $$.formals = $3.formals}
    ;

formals :
  /* empty */
      {$$.formals = []}
  | formal
      {$$.formals = [$1.env_binding]}
  | formal ’,’ formals
      {$$.formals = $1.env_binding @
        $3.formals}
  ;
```

SIL Rules, cont’d

```
formal :
  var ’:’ type
    {$$.env_binding =
      ($1.name, $3.type)}
  ;
```

SIL Rules, cont’d

```
procbody :
  BEGIN stmts END
    {$2.env = $$.env
     $$.type = $2.type
     $$.fun_body = functionize(
         $2,(env*store),store’)}
  ;
```

SIL Rules, cont’d

```
stmts :
  stmt ’;’
    {$1.env = $$.env
     $1.store = $$.store
     $$.type = $1.type
     $$.store’ = $1.store’}
  | stmt ’;’ stmts
    {$1.env = $3.env = $$.env
     $$.type =
       if $1.type = "OK" and
          $3.type = "OK"
       then "OK" else error_type
     $1.store = $$.store
     $3.store = $1.store’
     $$.store’ = $3.store’}
  ;
```

SIL Rules, cont’d

```
stmt :
  /* empty */
  | var ’:=’ expr
    {$$.type =
       if #2(assoc($1.name, $$.env))
          = $3.type
       then "OK" else error_type
     $3.store = $$.store
```

SIL Rules (stmt), cont’d

```
 $$.store’ =
   if (length($$.store) > 1) and
     assoc($1.name, hd($$.store))
   then
     reassign($1,name, $3.value,
       hd($$.store)) @ tl(store)
   else if assoc($1.name,
           last($$.store))
   then butlast(store) @ reassign(
     $1.name, $3.value, last($$.store))
   else
     butlast(store) @ assign(
      $1.name, $3.value, last($$.store))
```

SIL Rules (stmt), cont’d

```
| IDENTIFIER ’(’ actuals ’)’
    {$$.type = if chk_apply(
       $1,name, $3.types, $$.env)
     $$.store’ = tl(apply(
       $1.name, $3.values, $$.env,
         $$.store))}
```

SIL Rules (stmt), cont’d

```
| IF expr THEN stmts ENDIF
  {$2.env = $4.env = $$.env
   $$.type =
     if $2.type = "boolean"
     then $4.type else error_type
    (* NOTE WEAKNESS HERE *)
   $4.store = $$.store
   $$.store’ =
     if $2.value
     then $4.store’ else $$.store
  }
```

SIL Rules (stmt), cont’d

```
  | IF expr THEN stmts ELSE stmts ENDIF
    {$2.env = $4.env = $6.env = $$.env
     $$.type =
         if $2.type = "boolean"
         then
           if $4.type = "OK" and
             $6.type = "OK
           then "OK"
           else error_type
            (* NOTE WEAKNESS HERE *)
     $4.store = $6.store = $$.store
     $$.store’ = if $2.value then $4.sto
    }
  ;
```

SIL Rules, cont'd

```
expression :
    number
            {$$.type = $1.type
             $$.store' = $$.store
             $$.value = $1.value}
    | char
            {$$.type = $1.type
             $$.store' = $$.store
             $$.value = $1.value}
    | bool
            {$$.type = $1.type
             $$.store' = $$.store
             $$.value = $1.value}
    | var
            {$$.type =
                 if assoc($1.name, $$.en
                     #2(assoc($1.name, $
                 else
```

```
                        error_type
                $$.store’ = $$.store
                $$.value =
                    if (length($$.store) >
                           assoc($1.name, h
                       #2(assoc($1.name, hd
                    else if assoc($1.name,
                       #2(assoc($1.name, la
                    else
                       error_value}
   | IDENTIFIER ’(’ actuals ’)’
               {$3.env = $$.env
                $3.store = $$.store
                $$.type = chk_apply($1,name
                $$.store’ = tl(apply(
                    $1.name, $3.values, $$.
                $$.value = last(hd(apply(
                    $1.name, $3.values, $$.
    | expr rel_op expr      %prec ’<’
               {$1.env = $3.env = $$.env
                $$.type =
```

```
                    if ($1.type = $2.type)
                    then $1.type
                    else error_type
                $1.store = $$.store
                $3.store = $1.store'    (* N
                $$.store' = $3.store '
                $$.value = $2.op_fun($1.val
      | expr add_op expr      %prec '+'
               {$1.env = $3.env = $$.env
                $$.type =
                    if ($1.type = $2.type)
                      (($1.type = "real") o
                    then $1.type
                    else error_type
                $1.store = $$.store
                $3.store = $1.store'
                $$.store' = $3.store '
                $$.value = $2.op_fun($1.val
      | expr mult_op expr     %prec '*'
               {$1.env = $3.env = $$.env
                $$.type =
```

```
                   if ($1.type = $2.type)
                      (($1.type = "real")
                   then $1.type
                   else error_type
               $1.store = $$.store
               $3.store = $1.store’
               $$.store’ = $3.store ’
               $$.value = $2.op_fun($1.val
      | ’(’ expr ’)’
              {$2.env = $$.env
               $$.type = $2.type
               $2.store = $$.store
               $$.store’ = $2.store’
               $$.value = $2.value}
      ;
```

SIL Rules, cont'd

```
add_op :
    '+'    {$$.op_fun = $1.op_fun}
    | '-'    {$$.op_fun = $1.op_fun}
    | OR     {$$.op_fun = $1.op_fun}
                /* NOTE: The lexer provid
    ;
```

SIL Rules, cont’d

```
mult_op :
    '*'   {$$.op_fun = $1.op_fun}
    | '/'   {$$.op_fun = $1.op_fun}
    | AND   {$$.op_fun = $1.op_fun}
    ;
```

SIL Rules, cont'd

```
rel_op :
    '<'   {$$.op_fun = $1.op_fun}
    | '>'   {$$.op_fun = $1.op_fun}
    | '='   {$$.op_fun = $1.op_fun}
    | '<='  {$$.op_fun = $1.op_fun}
    | '>='  {$$.op_fun = $1.op_fun}
    | '<>'  {$$.op_fun = $1.op_fun}
    ;
```

SIL Rules, cont’d

```
actuals :
    /* empty */
            {$$.types = []
             $$.store’ = $$.store
             $$.values = []}
    | actual
            {$1.env = $$.env
             $1.store = $$.store
             $$.types = [$1.type]
             $$.store’ = $1.store’
             $$.values = [$1.value]}
    | actual ’,’ actuals
            {$1.env = $3.env = $$.env
             $1.store = $$.store
             $3.store = $1.store’   /* N
             $$.store’ = $3.store’
             $$.values = $1.value @ $3.v
    ;
```

SIL Rules, cont'd

```
actual :
    expr
            {$$.type = $1.type
             $$.store' = $1.store'
             $$.value = $1.value}
    ;
```

SIL Rules, cont’d

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

SIL Rules, cont’d

```
real :
    REALVAL
            {$$.type = "real"
             $$.value = $1.value}
               /* The lexer provides rea
    ;
```

SIL Rules, cont'd

```
integer :
    INTEGERVAL
            {$$.type = "integer"
             $$.value = $1.value}
               /* The lexer provides int
    ;
```

SIL Rules, cont'd

```
char :
    CHARVAL
            {$$.type = "char"
             $$.value = $1.value}
               /* The lexer provides cha
    ;
```

SIL Rules, cont’d

```
bool :
    BOOLVAL
            {$$.type = "boolean"
             $$.value = $1.value}
               /* The lexer provides boo
    ;
```