CHAPTER 9

RIGOROUS DEVELOPMENT OF INTERPRETERS AND COMPILERS

This chapter again tackles the problem of compiler development. The language considered is SAL, a Simple Applicative Language, which is not that simple: it handles functions as values - including such delivered out of their defining scope. Thus it illustrates the so-called FUNARG property [Moses 70a, Weizenbaum 68a]. The intermediate steps use imperative constructs of the meta-language (e.g. declarations) which are described in the Glossary of Notation. The example is used to illustrate the step to compiling algorithms and uses 'attribute semantics'. (The chapter is a rewritten, abbreviated version of [Bjørner 77b].)

CONTENTS

9.1 INTRODUCTION

Starting with a denotational semantics definition of a simple applicative language, SAL, we systematically develop the specifications of a compiler for SAL. We do so by presenting, in a unifying framework -- and steps of increasing concretization -- the commonly known semantics definition styles of the 1960's and 1970's:

i. denotational semantics
ii. first-order functional semantics
iii. abstract state machine operational semantics
iv. concrete state machine operational semantics
v. attribute semantics.

The first four semantics styles are employed in the definition of an interpetive semantics, whilst the fifth style is engaged in the final description of a compiling algorithm. The target machine for which the compiler is to generate code is likewise interpretively defined.

The double aim of this chapter is to advocate a different approach to the teaching of compiler design; and to illustrate that the spectrum of semantics definition methods of the 1960s fit into a development hierarchy. The main content of the chapter is seen as the exemplification of a disciplined software development methodology, especially as applicable to programming language design and compiler development, and the demonstration of its feasibility. The implied, derived and constituent aspects of the chapter are then these: the design of a hierarchy of meta-languages for expressing levels of abstraction and concretization; and the placement in a proper context, blending and exploitation of a number of seemingly diverse software techniques. These latter include the conscious choice and/or mixture of levels of representational and operational abstraction, configurational (bottom-up) and hierarchical (top-down) abstraction, and functional (applicative) versus state (imperative) programming.

We believe, seemingly contrary to all textbooks on compiler design, that the very initial stages of any compiler development must concentrate first on precise descriptions of the source and the target languages; to be followed by a precise description of the compiling algorithm. That is: of the compiler's input/output relation: source program texts into target code sequences. We also believe that an activity such as the one whose

initial steps have been outlined above, can be meaningfully embedded within a more generally applicable software development methodology.

The borderline between modelling the source language abstractly for purposes of language design and compiler and program development are these: the language designer experiments with different models in attempts to understand, discover, purify, generalize and simplify language constructs. The compiler developer uses the final abstraction document as a basis for implementation of the compiler. And the source language programmer refers to the mathematical semantics definition when proving correctness of source programs. In this paper we shall exemplify only the compiler developer's view.

9.2 INFORMAL DESCRIPTION OF SAL

Syntax:

SAL is a simple applicative language. Its programs are expressions. There are eight expression categories:

*Const*ants	k
*Var*iables	id
Infix expressions	e1 + e2
*Cond*itional expressions	if et then ec else ea
Simple *Let* Blocks	(let id = ed in eb)
*Rec*ursive Functions	(letrec g(id) = ed in eb)
*Lamb*da Functions	λid.ed
*Appl*ications	ef(ea)

(Most of our elaboration functions, incidentally, will be expressed in a simple language like SAL.) Blocks with multiple definitions can be "mimicked" by multiply nested simple (*Let*) blocks. Multiple, mutually recursive functions, however, cannot be explicitly defined other than through the use of formal function arguments.

Data Types:

Constants stand for Natural numbers, Booleans, etc. The infix operators are then the usual ones: ADDition, SUBtraction, AND etc.

Comment:

SAL may seem awfully trivial to those who are used to programming with an ample supply and type variety of assignable variables -- but its realization illustrates most of the more intricate aspects of interpreter, that is runtime code, and compiler design. The main reason for this should be seen in SALs ability to yield FUNction VALues out of their defining scope (that is the so-called FUNARG property [Moses 70a, Weizenbaum 68a]). In addition, our development concentrates on implementing the block-structure and function invocation aspects.

Semantics:

SAL programs express only three kinds of VALues: Natural numbers, truth valued Booleans, and FUNction VALues, that is objects which are functions from VALues to VALues, these again including FUNctions, etc.. The DENotation, that is VALue, of a variable identifier, 'id', is that of the possibly recursively defined) defining expression: 'ed' (respectively: 'Yλg.λid.ed') of the lexicographically youngest incarnation, that is the "outwardgoing" statically closest containing block. Y is the fixed point finding function which when applied to 'λg.λid.ed' yields the "smallest" solution to the equation: 'g(id)=ed', in which 'g' occurs free in 'ed'. Infix and conditional expression VALues are as you expect them to be. The VALue of a block is that of the expression body, 'eb', in which all free occurences of the 'id' of a let, respectively the 'g' of a letrec, block header definition have been replaced (or: substituted) by their VALues. That is: 'ed' is evaluated in an environment, *env'*, which is exactly that extension of the block-embracing environment, *env*, which binds 'id' (respectively 'g') to its VALue, and otherwise binds as *env*. The VALue of a lambda-expression, 'λid.ed', is the FUNction of 'id' that 'ed' denotes in the environment in which it is first encountered, that is defined. Finally: the VALue of an application, 'ef(ea)', is the result of applying the FUNction VALue that 'ef' denotes to the VALue denoted by 'ea'.

9.3 INTERPRETIVE SEMANTICS DEFINITIONS

Four styles will be given. In the first definition we express the semantics of SAL in terms of mathematical functions. Thus the semantics of a compound syntactic object is expressed as the (homomorphic) function

that is as functional composition) of the semantics of the individual, proper components. The denoted functions are themselves expressed in terms of so-called semantic domains, and these are again functional. The remaining definitions are increasingly more 'computational', that is can best be understood as specifying sequences of computations given an input, that is an initial binding of variables to their meaning.

The last, fourth, interpretive definition "unzips" user-defined functions by permitting a compiletime macro-expansion of the definition, pre-processing SAL program-defined functions into label/goto "bracketed" meta-language texts, and calls of these functions into (branch and link-like) gotos to such texts. The principles of properly saving, updating (that is "setting-up") and restoring (that is "taking-down" & "re-installing") calling and defining environments form a more detailed version than any of the preceding definitions, and of otherwise published accounts of this so-called static (environmentally preceding) and dynamic (call) activation chain mechanisms.

9.3.1 Denotational Semantics (I)

Without much further ado we now present the first in a series of seven specifications of SAL (I-IV,VI-VIII).

I.1 Syntactic Domains

(1) *Prog* = *Expr*

(2) *Expr* = *Const* | *Var* | *Infix* | *Cond* | *Let* | *Rec* | *Lamb* | *Appl*

(3) *Const* :: *Int*

(4) *Var* :: *Id*

(5) *Infix* :: *Expr Op Expr*

(6) *Cond* :: *Expr Expr Expr*

(7) *Let* :: *Id Expr Expr*

(8) *Rec* :: *Id Lamb Expr*

(9) *Lamb* :: *Id Expr*

(10) *Appl* :: *Expr Expr*

(11) *Id* ⊂ *Token*

(12) *Op* = ADD | SUB | AND | ...

I.2 Semantic Domains

(13) *ENV* = *Id* $\overrightarrow{m}$ *VAL*

(14) *VAL* = *Int* | *Bool* | *FUN*

(15) *FUN* = *VAL* $\rightarrow$ *VAL*

I.3 Elaboration Functions

```
(16) type: eval-prog:  Prog → VAL
(17) type: eval-expr:  Expr → (ENV → VAL)
(18) type: eval-fun:   Lamb → (ENV → FUN)

16. eval-prog[e] ≜ eval-expr[e]([])

18. eval-fun[mk-Lamb(id,e)]env ≜ λa.eval-expr[e](env+[id ↦ a])

17. eval-expr[e]env ≜
 .1 cases e: mk-Const(k)        → k,
 .2          mk-Var(id)         → env(id),
 .3          mk-Infix(e1,o,e2)  → (let v1 = eval-expr[e1]env,
 .4                                    v2 = eval-expr[e2]env in
 .5                                cases o: ADD→v1+v2,SUB→v1-v2,...)),
 .6          mk-Cond(t,c,a)     → if eval-expr[t]env
 .7                                  then eval-expr[c]env
 .8                                  else eval-expr[a]env,
 .9          mk-Let(id,d,b)     → (let env'=env+[id ↦ eval-expr[d]env] in
 .10                               eval-expr[b]env'),
 .11         mk-Rec(g,d,b)      → (let env'=env+[g ↦ eval-fun[d]env'] in
 .12                               eval-expr[b]env'),
 .13         mk-Lamb( , )       → eval-fun[e]env,
 .14         mk-Appl(f,a)       → (let fun = eval-expr[f]env,
 .15                                   val = eval-expr[a]env      in
 .16                               if is-FUN(fun)
 .17                                  then fun(val) else undefined
```

9.3.2 First-Order Applicative Semantics (II)

By a first-order applicative semantics definition we mean one whose semantic domains are non-functional, but which is still referentially transparent. Hence, if we were given, as a basis, a denota tional semantics we would have to object transform its functional components into such objects which by means of suitable "simulations" can mimic the essential aspects of the denotational definition. In the case of SAL two kinds of objects are to ve transformed: $ENV = Id \xrightarrow{m} VAL$ and, among VALues: $FUN = VAL \xrightarrow{\sim} VAL$. The former objects were constructed by means of expressions:

I.18. $env' = env + [id \mapsto a]$

I.17.9 $env' = env + [id \mapsto eval\text{-}expr[d]env]$

I.17.11 $env' = env + [g \mapsto eval\text{-}fun[d]env']$

The latter objects were denoted by an expression basically of the lambda form:

I.18. $\lambda a.(eval\text{-}expr[e](env + [id \mapsto a]))$

We shall not motivate the transformation choices further (see [Reynolds 72a]), nor state general derivation principles, but rather present the transformed objects as "faits-accomplis": *ENV* objects, which are MAPs ($\vec{m}$), as *ENV1* objects of the tuple type, with extensions accomplished in terms of concatenations (^), and functional application (*()*) as directed, linear searches (*look-up1*). The mathematical functions, *fun*, denoted by lambda-expressions are then realized as so-called *closures* -- these are 'passive' structures, which pairs the expression, *d*, to be evaluated, with the defining environment, *env'*, so that when *fun* is to be applied, *fun(val)*, then a simulation of *clos* with the transformed counterpart, *arg*, of *val*, is performed: *apply1(clos,arg)*.

Instead of now presenting the more concrete, first-order functional elaboration functions we first present arguments for why we believe that our choices will do the job. Those arguments are stated as *retrieve functions*, *retr-ENV* and *retr-VAL*, which apply to the transformed objects and yield the more abstract "ancestors" from which they were derived. We next observe that the definition is still functional, as was the denotational. All arguments are explicit, there is no reference to assignable/declared variables. And we finally note that we cannot, given a specific expression, *e*, 'stick' it into the *m1-eval-expr* (together with an initial, say empty environment) and by macro-substitution eliminate all references to *m1-eval-expr*. The reason for this "failure" will be seen in our "stacking" closures whose subsequent application requires *m1-eval-expr*.

II.1 Syntactic Domains - as in I.1

II.2 Semantic Domains

(1) *ENV1* = *IdVal**

(2) *IdVAL* = *SIMP* | *REC*

(3) *SIMP* :: *Id* *VAL1*

(4) *REC* :: *Id* *Lamb*

(5) *VAL1* = *Int* | *Bool* | *CLOS*

(6) *CLOS* :: *Lamb* *ENV1*

II.2.1 Retrieve Functions

```
(7) type: retr-ENV:  ENV1 → ENV
(8) type: retr-VAL:  VAL1 → VAL

7.0 retr-ENV(env1) Δ
 .1  if env1=<>
 .2   then [],
 .3   else
 .4    (let env = retr-ENV(tl env1) in
 .5     cases hd env1:
 .6       mk-SIMP(id,val1) → env + [id ↦ retr-VAL(val1)],
 .7       mk-REC(g,d)      → (let env' = env + [g ↦ eval-fun[d]env'] in
 .8                           env')))

8.0 retr-VAL(val1) Δ
 .1  cases val1: (mk-CLOS(l,env1) → eval-fun[l](retr-ENV(env1)),
 .2               T               → val1)
```

II.2.2 Auxiliary Function

```
(9) type: look-up1:  Id × ENV1 → VAL1

9.0 look-up1(id,env1) Δ
 .1  if env1=<>
 .2     then undefined
 .3     else cases hd env1: mk-SIMP(id,val1) → val1,
 .4                         mk-REC(id,lamb)  → mk-CLOS(lamb,env1),
 .5                         T                → look-up1(id,tl env1)
```

II.3 Elaboration Functions

```
(10) type: m1-eval-prog:  Prog          ⇸ VAL1
(11) type: m1-eval-expr:  Expr → (ENV1 ⇸ VAL1)
(12) type: apply1:        CLOS × VAL1   ⇸ VAL1

10.0 m1-eval-prog[e] Δ m1-eval-expr[e](<>)

11.0 m1-eval-expr[e](env1) Δ
  .1  cases e: mk-Const(k)  → k,
  .2           mk-Var(id)   → look-up1(id,env1),
```

```
.3    mk-Infix(e1,o,e2) → (let v1 = m1-eval-expr[e1](env1),
.4                             v2 = m1-eval-expr[e2](env1)      in
.5                          cases o:
.6                            ADD → v1+v2, SUB → v1-v2,...)),
.7    mk-Cond(t,c,a)    → if m1-eval-expr[t](env1)
.8                          then m1-eval-expr[c](env1)
.9                          else m1-eval-expr[a](env1),
.10   mk-Let(id,d,b)    → (let v      = m1-eval-expr[d](env1)   in
.11                         let env1' = <mk-SIMP(id,v)>^env1    in
.12                         m1-eval-expr[b](env1')),
.13   mk-Rec(g,d,b)     → (let env1' = <mk-REC(g,d)>^env1      in
.14                         m1-eval-expr[b](env1')),
.15   mk-Lamb(,)        → mk-CLOS(e,env1),
.16   mk-Appl(f,a)      → (let clos = m1-eval-expr[f](env1),
.17                             arg  = m1-eval-expr[a](env1)    in
.18                         apply1(clos,arg))
```

```
12.0 apply1(clos,arg) ≜
  .1  cases clos:
  .2   mk-CLOS(mk-Lamb(id,d),ρ1) → (let ρ1'=<mk-SIMP(id,arg)>^ρ1 in
  .3                                  m1-eval-expr[d](ρ1')),
  .4   T                         → undefined
```

9.3.3 Abstract State Machine Semantics (III)

By an abstract state machine semantics we understand a definition which typically employs (globally) declared variables of abstract, possibly higher-level, type. It expresses the semantics (not in terms of applicatively defined, "grand" transformations on this state, but) in terms of statement sequences denoting a computational process of individual, "smaller" state transformations.

In the SAL case we choose to map the semantic *ENV1* arguments onto a globally declared variable, *env2*, thereby removing these arguments from the elaboration function references. By doing so we must additionally mimic the meta-language's own recursion capability which is exploited e.g. in lines II.11.3-4,7-9, ... Thus the type of *env2* is to become a stack of stacks, that is: *ENV2=ENV1**, where *ENV1=IdVal**. Each env2 element is that stack of *Id*'s and their values, which when *looked-up* properly (cf. *retr-ENV*) reflects the bindings of the so-called "lexicographically youngest incarnations" of each identifier in the static scope, that is:

in going outwards from the identifier use through embracing blocks towards the outermost program expression level. As long as no let or rec defined function is being *Applied*, the 'env2' will contain exactly one *ENV1* element. As soon as a defined function is *Applied*, the calling environment is dumped on the 'env2' stack. On its top is pushed the *ENV1* environment current when the function was defined.

In addition we choose to mechanize the recursive stacking of temporaries (e.g. II.11.3-4, 10, 16-17) by means of a global stack, *STK*. We could have merged *STK* into *ENV2*, but decide not to at present. Hence this abstract machine definition also requires further decomposition of the *look-up1* operation. As before, we state our beliefs as to why we think the present development is on a right track, by presenting *retrieve* functions.

The abstract state machine semantics definition is said to be operational, or to be a mechanical semantics definition, since it specifies the meaning of SAL by describing the operation of a machine effecting the computation of the desired value. Such definitions rather directly suggests, or are, realizations. They do not possess, or involve, implicit, implementation language processor controlled, but explicit state machine semantics definer determined allocation and freeing. We refer to: ^, respectively *tl*. The allocation and freeing is of otherwise recursively nested (that is stacked) objects. The definition, however, still requires the presence, at run-time, of *m2-eval-expr* (III.23.8). It still cannot be completely factored out of the definition for any given, non-trivial expression. Thus there still cannot be an exhaustive, macro-substitution process which completely eliminates the interpretive nature of the definition. The reason is as before: *CLOS*ures are triplets of a function definition bound variable, *id*, a function 'body', *d*, and the recursive, defining environment, *env2'*. Together they represent, but are not, the function, *fun* (I.18). It must instead be mimicked; hence the required presence of *m2-eval-expr*.

III.1 Syntactic Domains -- as in I.1

III.2 Semantic Domains

(1) *ENV2* = *ENV1**
(2) *ENV1* = *IdVal**
(3) *IdVal* = *SIMP* | *REC*
(4) *SIMP* :: *Id* *VAL1*
(5) *REC* :: *Id* *Lamb*
(6) *VAL1* = *Int* | *Bool* | *CLOS*
(7) *CLOS* :: *Lamb* *ENV1*
(8) *STK* = *VAL1**

(9) $\Sigma = (\text{env2} \xrightarrow{m} ENV2) \cup (\text{stk} \xrightarrow{m} STK)$

III.2.1 State Initialization

(10) *dcl* env2 := <<>> *type* *ENV2*,
(11) stk := <> *type* *STK*;

III.2.2 Retrieve Functions

(12) *type*: *retr-ENV1*: $\Sigma \to ENV1$ *retr-ENV1*() $\triangleq$ *hd* $\frown$ env2
(13) *type*: *retr-VAL1*: $\Sigma \to VAL1$ *retr-VAL1*() $\triangleq$ *hd* $\frown$ stk

III.2.3 Auxiliary Function

(14) *type*: *look-up2*: $Id \to (\Sigma \xrightarrow{\sim} \Sigma)$

```
14.0 look-up2(id) ≜
  .1   (trap exit() with I in
  .2    for j=1 to len hd ⌢ env2 do
  .3      cases hd ⌢ env2[j]:
  .4        mk-SIMP(id,val2)
  .5             → (stk := <val2> ^ ⌢ stk;
  .6                exit),
  .7        mk-REC(id,e)
  .8             → (def env2' : <(hd ⌢ env2)[k] | j ≤ k ≤ len hd ⌢ env2>;
  .9                stk := <mk-CLOS(e,env2')> ^ ⌢ stk;
  .10               exit)
  .11       T    → I);
  .12   error)
```

III.3 Elaboration Functions

(15) *type*: *m2-eval-prog*: *Prog* $\to (\Sigma \xrightarrow{\sim} \Sigma\ VAL1)$
(16) *type*: *m2-eval-expr*: *Expr* $\to (\Sigma \xrightarrow{\sim} \Sigma)$

```
15.0 m2-eval-prog[e]                ≜ (env2) := <<>>;
  .1                                    m2-eval-expr[e];
  .2                                    env2 := tl ⌢ env2;
  .3                                    return(hd ⌢ stk))

16.0 m2-eval-Const[mk-Const(k)]     ≜ stk := <k>^⌢ stk
```

```
17.0 m2-eval-Var[mk-Var(id)]                ≜ look-up2(id)

18.0 m2-eval-Infix[mk-Infix(e1,o,e2)] ≜
  .1       (m2-eval-expr[e1];
  .2        m2-eval-expr[e2];
  .3        stk := <hd tl c stk  cases o: ADD → +, SUB → -,...  hd c stk>
  .4              ^ tl tl c stk)

19.0 m2-eval-Cond[mk-Cond(t,c,a)]           ≜ (m2-eval-expr[t];
  .1                                          def b: hd c stk;
  .2                                          stk := tl c stk;
  .3                                          if b then m2-eval-expr[c]
  .4                                               else m2-eval-expr[a])

20.0 m2-eval-Lamb[e] ≜  stk := <mk-CLOS(e,hd c env2)> ^ c stk

21.0 m2-eval-Let[mk-Let(id,d,b)]            ≜
  .1       (m2-eval-expr[d];
  .2        env2 := <<mk-SIMP(id,hd c stk)> ^ hd c env2> ^ tl c env2;
  .3        stk  := tl c stk;
  .4        m2-eval-expr[b];
  .5        env2 := <tl hd c env2> ^ tl c env2)

22.0 m2-eval-Rec[mk-Rec(g,d,b)]             ≜
  .1       (env2 := <<mk-REC(id,d)> ^ hd c env2> ^ tl c env2;
  .2        m2-eval-expr[b];
  .3        env2 := <tl hd c env2> ^ tl c env2)

23.0 m2-eval-Appl[mk-Appl(f,a)] ≜
  .1       (m2-eval-expr[a];
  .2        m2-eval-expr[f];
  .3        if is-CLOS(hd c stk)
  .4           then (def mk-CLOS(mk-Lamb(id,e'),env2') : hd c stk;
  .5                 env2 := <<mk-SIMP(id,hd tl c stk)>^env2'>^c env2;
  .6                 stk  := tl tl c stk;
  .7                 m2-eval-expr[e'];
  .8                 env2 := tl c env2)
  .9           else error)
```

9.3.4 Concrete State Machine Semantics (IV)

By a concrete state machine semantics we understand a definition which again exploits globally declared variables, but now of more concrete, efficiently realizable type. We shall in particular mean such forms which model, or rather closely exhibit, the actual run-time structure of for example such objects as activation stacks, but such that the definition is still interpretable within, at this time, an extended meta-language. It is observed that the borderline between the definition styles is smooth, and thus that too rigid delineations serve no purpose. In the abstract state machine semantics of SAL we observe a number of storagewise inefficient object representations; these are caused almost exclusively by our choice to stay with the CLOSure representation of FUNctions as first derived in sect. 9.3.2. Closures "drag" along with them, not only the function body text, but also the entire defining environment. This generally results in extensive duplication of dynamic scope information being kept "stored" in *ENV2*. The basic object transformation objective therefore, of this development step, is now to keep only nonredundant environment information in the transformed activation stack. We shall achieve this by "folding" the *ENV2* stack of *ENV1* stacks "back into" a tree structured activation stack (*STG*). Each "path" from a leaf to the root signifies a chain of dynamically preceding activations, with one of these chains signifying the current, all others those of defining, environment chains of FUNARG functions. Each chain is statically and dynamically linked, corresponding to the subchain of environmentally preceding, lexicographically youngest, that is most recent, incarnations of statically embracing blocks; respectively the complete chain of dynamically (call/invocation) preceding activations. Our definition thus entails a complete, self-contained description of a commonly used variant of the so-called DISPLAY variable referencing scheme first attributed to Dijkstra [Dijkstra 62a]. We can, however, only succeed in achieving this realization of activations if, at the same time we refine *CLOSures* into pairs of resulting program *label* points, *lfct*, and defining environment activation stack pointers, *cp*. From *lfct* we are able to *retrieve* the *Lambda* expression, and from *cp* we are able to *retrieve* the defining *environment*.

Compiler-Compilers:

To realize this goal we also, in this step, refine *CLOSures* by macro-expansion compilation of SAL texts, *e*, into extended meta-language texts.

texts. It is thus we have chosen here to introduce, somewhat belatedly, but - we think - in an appropriate context, the issue of viewing a meta-language expressed definition of some source language construct as specifying a compilation of source language texts into meta-language texts. By a meta-language, macro-substitution, compiled (interpretive) semantics we basically understand a definition in the meta-language not containing any references to specifier defined elaboration functions. We shall, however, widen the above to admit forms, which contain such references, but where these now are to be thought of as references to elaboration macros, hence implying a pre-processing stage, called compiling, prior to interpretation of 'pure' meta-text, that is metatext free from references to specifier defined functions. We are given input source texts in the form of arguments to elaboration functions. To achieve an extended meta-language definition, which can be so macro-expanded, recursive definitions of objects (like for example *env'* I.17.9) and functions must be eliminated. We do so either by taking their fixed points, or by "unzipping" them into mechanical constructions. Taking fixed points, for example results in:

$$\underline{let}\ env' = Y\lambda\rho.(env + [id \rightarrow eval\text{-}expr(d)\rho])$$

but that doesn't help us very much when we come to actual, effective realizations on computers - it is, or may be, beautiful in theory, but "costly" in practice. Even though computers may be claimed to possess fixed point finding instructions, Y, they would have to be general enough to cater for the most complex case. Instead we unravel each individual use of recursion separately, and so-far by hand. In the case of *env'* by providing suitable stacks, pointer initializations and manipulations. The guiding principle being: to derive, from the more abstract definition, to each occurrence of an otherwise recursive definition a most fitting, efficient and economical realization. In the next five subsections (A-B-C-D-E) we now go into a characterization of the resulting definition at this stage. Again we present it as a "fait-accompli", leaving to other treatments the formulation of (and partial, theoretical support for) the general derivation techniques applied.

The definition represents two intertwined efforts: the further concretization of run-time objects, here the *ENV2* stack into the Σ complex, and the further decomposition of elaboration function definitions so that we can come to the point where references such as *m3-int-Expr*, can be successively eliminated.

A: The Environmentally Preceding Activation *(EPA)* & Variable Referencing Scheme, and the Run-Time State

The *ENV2* and *STK* of III is merged into the separately allocated *DSA*s (Dynamic Storage Areas) of *STG*. These are chained together: dynamic chains by *CP* (for: Calling Pointer), lexicographic chains to (defining) youngest incarnations by *EP* (for: Environment Pointer). The exact functioning of this *EPA* scheme is precisely described by the formulae. Hence it will not be informally described here. Our objective in presenting the formulae (IV.1-IV.32) is twofold: (1) to indicate a stepwise refinement process which leads to their derivation and the possibility now of a correctness proof with respect to a far shorter definition, and (2) to show that, even when starting with the concrete, which most textbooks unfortunately still do, and then invariably only very incompletely, one can indeed achieve a complete yet terse formulation.

B: Macro-Expansion

A conditional (*Cond*) expression, for example, results in all of the text corresponding to IV.26.1,4-5 being generated first, in a pre-processing 'compile' stage. A simple *Let* defining block expression, for example, results in all of the text corresponding to IV.27.1,.3 being expanded before any elaboration. Etcetera. Thus lines IV.21.1, -28.5,.12, -31.4 etc., do not denote themselves, that is run-time references to *m3-int-Expr*, but the text resulting from similar expansions. One may choose to do likewise for the auxiliary functions *Pop* and *Push*, or one may wish to keep these as standard run-time routines.

C: Realization of *CLOS*ures

Note the *Rec* or *Lamb* cases: 'letrec g(id)=d in b' respectively: '$\lambda id.id$'. Upon evaluation of a *Rec* or a *Lamb* their defined function bodies, *d*, are not elaborated (until actually *Applied*). Since we have decided to macro-expand these texts "in-line" with the text in which they were defined, and since we are not to execute this text when otherwise elaborating the two definition cases, we shall (i) *label* their expansions, (ii) *label* the text immediately following these expansions, (iii) precede the expansion with a (meta-language) *GOTO* around the thus expanded text, (iv) and terminate the expanded text itself with a *GOTO* intended to return to the caller, who, it is expected, "dropped" a suitable return address in a global 'Ra' branch label register before *GO*ing *TO* the *label* of the expanded

function text. All this is "performed" in functions IV.30, respectively IV.29. So what is left in the *EPA* of the former *CLOS*ures? The answer is: just the "barebones". Enough to reconstruct (that is *retr*ieve) the *id*, the *d*, and their defining *env*ironment: the former two from *lab(fct)*, the *env*ironment from *c*'p' (IV.3.4, IV.30.8). Thus, in this definition, a function *CLOS*ure has been realized as a *FCT* pair: *(fct, ptr)*. This solution closely mirrors the way in which procedures are realized in actual programming language systems.

D: The Compiler State

We observe that *Labels* had to be generated for each *Lamb*, *Appl* and *Rec* (actually its *Lamb* part), and since we describe only once (in IV.30 and IV.29) what meta-language text to be generated, that is how to schematically elaborate these, we shall have to view the formulae (IV.21-31) as subject to (as already mentioned) a two stage process: the 'compiler' stage which macro-expands the SAL program into "pure" meta-text, and the 'interpreter' which executes the expanded text. Thus 11 lines of the formulae, namely those with lower case *let* and *def*, are 'executed' at compile-time, all *dict* (in *DICT*) objects are likewise compile-time computed, and all references to *m3-* functions are eliminated by the compile-time macrosubstitution process already mentioned. All upper-case *LET* and *DEF*s, are then to be executed at "run-time", that is in the interpreter stage. Thus the abstract compiler, whose "working behavior" will not be formalized in this paper, performs three actions: it generate labels; it computes, distributes and uses dictionaries; and it generates *META-IV* texts. Whereas in *ENV1* and *ENV2* *VAL*ues of *id*s were explicitly paired with these, in *DSA*s only the *VAL*ues are left, but in fixed positions *(VR)*. Consider any variable, *id*. It is defined at block depth *n*, and uniquely so. And it is used, for example at block depth *ln*, where: $0 \leq n \leq ln$. The *DICT* components serve exactly this singular purpose (at least in this sample definition): for all *id*s in some context, to map them into the static block depth, *n*, at which they were defined. Since the static chain also touches exactly the embracing blocks, *ln-n* denotes the number of levels one has to chain back to get to the *VAL*ue corresponding to *id* (IV.24.4). In fact, that is the whole, singular purpose of the static (*EP*) chain. Since it is furthermore observed (IV.24.1) that the only place *dict* is used, is in the compile stage, any reference to *dict* is seen also to be eliminated. Finally observe, that the unique label objects denoted by *lfct*, *lbyp* and *lret* shall be substituted into respective uses (IV.30.4,.8; IV.30.3,.7; IV.29.9,.11).

E: Execution

The result of of executing a SAL program is to be found on top of the temporary list (IV.19.2), about which we can assert a length of exactly one in line (IV.21.2)! So *m3-int-Expr* places (*m3-pushes*) the result of any expression elaboration on top of the current *DSA*'s *TL* -- with the working register, Ur, invariably holding this result too at the instance of *pushing*. A simple *Let Expr* is executed by first finding the *VAL*ue of the locally defined variable, *id*, in the environment in which the *Let* is encountered. Then a new activation is set up to elaborate the body, *b*, of the *Let*. Working register Ur is used to store the result temporarily while the activation is terminated, but not necessarily disposed off. The result is *pushed* on the *TL* of the invoking activation's *DSA*. Since the *VAL*ue so yielded might be a function which was "concocted" by the activation just left, and since that *FunCT*ion may depend on its locally defined *Var*iable *VAL*ues, we cannot, in general, dispose of the activation. This "story" then shall account for our use of the (--) dashed line around the reclamation of *ST*orage shown in (IV.31.9, IV.28.17). The yielded *FunCT*ion *VAL*ue would be (realized as) a pair: *mk-FCT(lfct,ptr)* where *ptr* is a *pointer* to that, or a contained, activation. This is again the FUNARG situation previously mentioned. By not disposing (IV.31.9) of the *DSA* we are later able to "reactivate" the *FunCT*ion defining activation. We leave it to the reader to "exercise" remaining aspects of the definition.

IV.1 Syntactic Domains -- as in I.1

IV.2 Semantic Domains

IV.2.1 Run-Time State Components

(1) $\Sigma t = (\mathrm{Stg} \xrightarrow{m} STG) \cup (\mathrm{Cp} \xrightarrow{m} [Ptr]) \cup (\mathrm{Ep} \xrightarrow{m} [Ptr]) \cup (\mathrm{Br} \xrightarrow{m} [Lbl]) \cup (\mathrm{Ra} \xrightarrow{m} [Lbl]) \cup (\mathrm{Ur} \xrightarrow{m} [VAL2]) \cup (\mathrm{Wr} \xrightarrow{m} [VAL2])$

(2,3,4) $P, EP, CP = [Ptr]$

(5,6) $BR, RA = [Lbl]$

(7) $STG = Ptr \xrightarrow{m} DSA$

(8,9) $Ptr, Lbl \subset Token$

(10) $DSA :: CP\ EP\ RA\ VR\ TL$

(11) $VR = [VAL2]$

(12) $TL = VAL2^*$

(13) $VAL2 = Int \mid Bool \mid FCT$

(14) $FCT :: BR\ \ EP$

Initial State

```
(15)   LET ptr    ∈ Ptr;
(16)   DCL Stg    := [ptr ↦ mk-DSA(nil,nil,nil,nil,<>)], type STG,
           Cp     := ptr                                type [Ptr],
           Ep     := ptr                                type [Ptr],
           Br     := nil                                type BR,
           Ra     := nil                                type RA,
           Ur,Wr  := nil                                type [VAL2];
```

IV.2.2 Compiler State

Global: (32) $\Sigma c = Ls \xrightarrow{m} Lbl\text{-}set$

(33) *dcl* Ls := {} *type Lbl-set;*

Local: (34) $LN = Nat$

(35) $DICT = Id \xrightarrow{m} Nat$

IV.3 Elaboration Functions

Function Types:

(21) *m3-int-Prog:* $Prog \to ((\Sigma c \to \Sigma c) \to (\Sigma t \to \Sigma t))$

(22) *m3-int-Expr:* $Expr \to ((DICT \times LN) \to ((\Sigma c \to \Sigma c) \to (\Sigma t \to \Sigma t)))$

(23) *m3-int-Const:* $Const \to ((DICT \times LN) \to (\Sigma t \to \Sigma t))$

(24) *m3-int-Var:* $Var \to ((DICT \times LN) \to ((\Sigma c \to \Sigma c) \to (\Sigma t \to \Sigma t)))$

(25) *m3-int-Infix:* $Infix \to ((DICT \times LN) \to ((\Sigma c \to \Sigma c) \to (\Sigma t \to \Sigma t)))$

(26) *m3-int-Cond:* $Cond \to ((DICT \times LN) \to ((\Sigma c \to \Sigma c) \to (\Sigma t \to \Sigma t)))$

(27) *m3-int-Let:* $Let \to ((DICT \times LN) \to ((\Sigma c \to \Sigma c) \to (\Sigma t \to \Sigma t)))$

(28) *m3-int-Rec:* $Rec \to ((DICT \times LN) \to ((\Sigma c \to \Sigma c) \to (\Sigma t \to \Sigma t)))$

(29) *m3-int-Appl:* $Appl \to ((DICT \times LN) \to ((\Sigma c \to \Sigma c) \to (\Sigma t \to \Sigma t)))$

(30) *m3-int-Lamb:* $Lamb \to ((DICT \times LN) \to ((\Sigma c \to \Sigma c) \to (\Sigma t \to \Sigma t)))$

(31) *m3-int-Block:* $Expr \to ((DICT \times LN) \to ((\Sigma c \to \Sigma c) \to (\Sigma t \to \Sigma t)))$

(18) *m3-Pop:* *ref VAL2* $\to (\Sigma t \to \Sigma t)$

(19) *m3-Push:* *ref VAL2* $\to (\Sigma t \to \Sigma t)$

(20) *make-Lbl:* $\to (\Sigma c \to \Sigma c)$

IV.3.1 Auxiliary Functions

Run-Time Functions:

```
18.0 m3-Pop(ref) ≜
  .1   (DEF mk-DSA(c,e,a,v,tl) : (c Stg)(c Cp);
  .2    Stg := c Stg + [c Cp ↦ mk-DSA(c,e,a,v,tl tl)];
  .3    ref := hd tl)

19.0 m3-Push(ref) ≜
  .1   (DEF mk-DSA(c,e,a,v,tl) : (c Stg)(c Cp);
  .2    Stg := c Stg + [c Cp ↦ mk-DSA(c,e,a,v,<c ref> ^ tl)])
```

Compile-Time Functions:

```
20.0 make-lbl() ≜
  .1   (def l∈Lbl\c Ls;
  .2    Ls := c Ls ∪ {l};
  .3    return(l))
```

IV.3.2 Compile/Execute Functions

```
21.0 m3-int-Prog[e] ≜
  .1   (m3-int-Expr[e]([],0);
  .2    Ur := hd(s-TL((c Stg)(c Cp)));
  .3    c Ur)

22.0 m3-int-Expr[e](dict,ln) ≜
  .1   is-Cons(e)  → m3-int-Const[e],
  .2   is-Var(e)   → m3-int-Var[e](dict,ln),
  .3   is-Infix(e) → m3-int-Infix[e](dict,ln),
  .4   is-Cond(e)  → m3-int-Cond[e](dict,ln),
  .5   is-Let(e)   → m3-int-Let[e](dict,ln),
  .6   is-Rec(e)   → m3-int-Rec[e](dict,ln),
  .7   is-Lamb(e)  → m3-int-Lamb[e](dict,ln),
  .8   is-Appl(e)  → m3-int-Appl[e](dict,ln)

23.0 m3-int-Const[mk-Const(k)] ≜
  .1   Ur := k; m3-Push(Ur)
```

```
24.0 m3-int-Var[mk-Var(id)](dict,ln) Δ
  .1   (let n = dict(id) in
  .2    Ep := c Cp;
  .3    FOR i=1 TO ln-n DO Ep := s-EP((c Stg)(c Ep));
  .4    Ur := s-VR((c Stg)(c Ep));
  .5    m3-Push( Ur);
  .6    Ep := c Cp)

25.0 m3-int-Infix[mk-Infix(e1,o,e2)](dict,ln) Δ
  .1   (m3-int-Expr[e1](dict,ln);
  .2    m3-int-Expr[e2](dict,ln);
  .3    m3-Pop(Ur);
  .4    m3-Pop(Wr);
  .5    Ur := c Ur  (cases o: ADD → +, SUB → -,...)  c Wr;
  .6    m3-Push(Ur))

26.0 m3-int-Cond[mk-Cond(t,c,a)](dict,ln) Δ
  .1   (m3-int-Expr[t](dict,ln);
  .2    m3-Pop(Ur);
  .3    IF c Ur
  .4       THEN m3-int-Expr[c](dict,ln)
  .5       ELSE m3-int-Expr[a](dict,ln))

27.0 m3-int-Let[mk-Let(id,d,b)](dict,ln) Δ
  .1   (m3-int-Expr[d](dict,ln);
  .2    m3-Pop(Ur);
  .3    m3-int-Block[b](dict + [id ↦ ln+1],ln+1))

30.0 m3-int-Lamb[mk-Lamb(id,d)](dict,ln) Δ
  .1   (def lfct : make-lbl(),
  .2        lbyp : make-lbl();
  .3    GOTO lbyp;
  .4    LAB(lfct);
  .5       m3-int-Block[d](dict + [id ↦ ln+1],ln+1);
  .6       GOTO c Ra;
  .7    LAB(lbyp):
  .8    Ur := mk-FCT(lfct,c Cp);
  .9    m3-Push(Ur))
```

```
31.0 m3-int-Block[b](dict,ln) ≜
  .1   (DEF ptr ∈ Ptr - dom c Stg;
  .2    Stg    := c Stg ∪ [ptr ↦ mk-DSA(c Cp,c Ep,c Ra,c Ur,<>)];
  .3    Cp,Ep := ptr;
  .4    m3-int-Expr[b](dict,ln);
  .5    m3-Pop(Ur);
  .6    Ep     := s-EP((c Stg)(c Cp));
  .7    Ra     := s-RA((c Stg)(c Cp));
  .8    Cp     := s-CP((c Stg)(c Cp));
  .9    ┌──────────────────────┐
        │ Stg := c Stg\{ptr};  │
        └──────────────────────┘
  .10   m3-Push(Ur))

28.0 m3-int-Rec[mk-Rec(g,mk-Lamb(id,d),b)](dict,ln) ≜
  .1   (def lfct: make-lbl(),
  .2         lbyp: make-lbl();
  .3    GOTO lbyp;
  .4    LAB(lfct):
  .5       m3-int-Block[d](dict + [g ↦ ln+1,id ↦ ln+2],ln+2);
  .6       GOTO c Ra;
  .7    LAB(lbyp):
  .8    DEF ptr∈Ptr\dom c Stg;
  .9    Ur     := mk-FCT(lfct,ptr);
  .10   Stg    := c Stg ∪ [ptr ↦ mk-DSA(c Cp,c Ep,c Ra,c Ur,<>)];
  .11   Cp,Ep := ptr;
  .12   m3-int-Expr[b](dict + [g ↦ ln+1],ln+1);
  .13   m3-Pop(Ur);
  .14   Ep := s-EP((c Stg)(c Cp));
  .15   Ra := s-RA((c Stg)(c Cp));
  .16   Cp := s-CP((c Stg)(c Cp));
  .17   ┌──────────────────────┐
        │ Stg := c Stg\{ptr};  │
        └──────────────────────┘
  .18   m3-Push(Ur))
```

```
32.0 m3-int-Appl[mk-Appl(f,a)](dict,ln) Δ
  .1  (def lret: make-lbl();
  .2   m3-int-Expr[a](dict,ln);
  .3   m3-int-Expr[f](dict,ln);
  .4   m3-Pop(Ur);
  .5   IF is-FCT(c Ur)
  .6      THEN (Br := s-BR(c Ur);
  .7            Ep := s-EP(c Ur);
  .8            m3-Pop(Ur);
  .9            Ra := lret;
  .10           GOTO c Br;
  .11           LAB(lret);
  .12           I)
  .13     ELSE ERROR)
```

9.4 COMPILING ALGORITHMS AND ATTRIBUTE SEMANTICS

In this section we shall arrive at a specification of SAL in terms of the combination of two separate definitions: a compiling algorithm which to any SAL construct specifies its translation, not into the meta-language, but into actual ("physically existing") machine code; and a suitably abstracted definition of the target machine architecture, that is its semantic domains (working registers, condition code, storage, input/output etc.) and instruction repertoire: formats and meaning. The structure of the section is as follows: in subsect. 9.4.1 we give the pair of definitions: target machine, TM, and the compiling algorithm form SAL to TM. The latter is directly derived from the last, concrete definition of SAL in sect. 9.3.4. In subsect. 9.4.2 we then restate the compiling algorithm of sect. 9.4.1, but now in terms more familiar to affecionados of attribute semantics. And in sect. 9.4.3 we give a similar attribute semantics definition of a compiling algorithm from SAL into TM. This latter algorithm is based on the separation of activation and temporary stacks first shown in the abstract machine of sect. 9.3.2. The purpose of showing the attribute semantics is to demonstrate, finally, how such are formally derivable from denotational semantics definitions. The reason why we state two independent attribute sematics definitions is to illustrate the distinctions between synthesized and inherited attributes, and their relation to questions about single- and multipass compilers. The latter will be discussed in sect. 9.5.

9.4.1 A Target Machine and a Compiling Algorithm (V-VI)

-- THE TARGET MACHINE, TM (V)

V.1 Syntactic Domains

(1) *Code* = *Ins**
(2) *Ins* = *Sim* | *St* | *Lim* | *Ld* | *Fct* | *Jmp* |
Cjp | *Mov* | *Adj* | *Pck* | *Unp* | *Pr* | ...
(3) *Sim* :: *Adr* (*Int* | *Bool* | ...)
(4) *St* :: *Adr* *Reg* *Nat1*
(5) *Lim* :: *Reg* (*Int* | *Bool* | *Lbl* | ...)
(6) *Ld* :: *Reg* *Nat1* *Adr*
(7) *Fct* :: *Reg* *Op* (*Reg* | *Adr*)
(8) *Op* = ADD | SUB | MPY | DIV | AND | OR | NOT | XOR | LOW | LEQ | EQ | NEQ | HI | HEQ | ...
(9) *Jmp* :: (*Lbl* | *Reg*)
(10) *Cjp* :: *Reg* *Cmp* (*Lbl* | ...)
(11) *Cmp* = TRUE | FALSE | ZERO | NOTFCT | ...
(12) *Pr* :: (*Reg* | *Quot*)
(13) *Adj* :: *Reg* *Int*
(14) *Mov* :: *Reg* *Reg*
(15) *Pck* :: *Reg* *Reg* *Reg*
(16) *Unp* :: *Reg* *Reg* *Reg*
(17) *Reg* = *Nat1*
(18) *Adr* :: *Base* *Displ*
(19) *Base* = *Reg*
(20) *Displ* = *Int*
(21) *Lbl* c *Token*

V.2 Semantic Domains

(22) Σm = (Stg $\xrightarrow{m}$ *STG*) ∪ (Reg $\xrightarrow{m}$ *REG*) ∪ (Out $\xrightarrow{m}$ *OUT*) ∪ ...
(23) *STG* = *LOC* $\xrightarrow{m}$ *VAL*
(24) *REG* = *Nat* $\xrightarrow{m}$ *VAL*
(25) *VAL* = *Int* | *Bool* | *Lbl* | *LOC* | *FCT*
(26) *LOC* = *Int*
(27) *FCT* :: *Lbl* *LOC*
(28) *OUT* = *VAL**

-- leaving a number of machine components undefined.

Global State Initialization:

```
(29) dcl Stg := [i → undefined | -2**s<i<2**s] type STG,
(30)     Reg := [i → undefined | -2**r<i<2**r] type REG;
```

V.3 (Micro-Program) Elaboration Functions

```
(31) type int-code: Code          → (Σm → Σm)
(32) type int-insl: Ins* (Nat1    → (Σm → Σm))
(33) type int-ins:  Ins           → (Σm → Σm)
(34) type eval-adr: Adr           → (Σm → Σm)

31.0 int-code[c] Δ
  .1   (fixe [lbl ↦ int-insl[c'](i) | (let c' = c^<mk-Lbl(ERROR)>  in
  .2                                   i = (Δj∈indsc')(c'[j]=lbl)]  in
  .3    int-insl[c](1))

32.0 int-insl[c](i) Δ
  .1    if i>lenc then I else (int-ins[c[i]];int-insl[c](i+1))

33.0 int-ins[ins] Δ
  .1   cases ins:
  .2     mk-Lbl(lbl)    → I,
  .3     mk-Sim(a,k)    → (def ea :  eval-adr[a];
  .4                        Stg    := c Stg + [ea ↦ k]),
  .5     mk-St(a,r,n)   → (def ea : eval-adr[a];
  .6                        for i=0 to n-1 do
  .7                            Stg := c Stg + [(ea+i) ↦ (c Reg)(r+i)]),
  .8     mk-Lim(r,k)    → Reg := c Reg + [r ↦ k],
  .9     mk-Ld(r,n,a)   → (def ea :  eval-adr[a];
  .10                       for i=0 to n-1 do
  .11                           Reg := c Reg + [(r+1) ↦ (c Stg)(ea+i)]),
  .12    mk-Fct(r,o,ar) →
  .13            (def v : is-Adr(ar) → (c Stg)(eval-adr[ar]),
  .14                     T          (c Reg)(ar);
  .15             cases o: ADD → Reg := c Reg + [r ↦ (c Reg)(r)+v],
  .16                      SUB → Reg := c Reg + [r ↦ (c Reg)(r)-v],
  .17                      ...
  .18                      HI  → Reg := c Reg + [r ↦ (c Reg)(r)>v],
  .19                      ...),
```

```
.20   mk-Jmp(l)        → exit(l),
.21   mk-Cjp(r,c,l)    → if cases c: TRUE    → (c Reg)(r),
.22                                 FALSE   → ¬(c Reg)(r),
.23                                 ...
.24                                 NOTFCT  → ¬is-FCT((c Reg)(r)),
.25                                 ...
.26                                 ZERO    → (c Reg)(r)=0,
.27                                 ...
.28                           then (is-Lbl(l) → exit(l),
.29                                 T         → exit((c Reg)(l)))
.30                           else I,
.31   mk-Adj(r,i)     → Reg := c Reg + [r ↦ (c Reg)(r)+i],
.32   mk-Mov(r1,r2)   → Reg := c Reg + [r1 ↦ (c Reg)(r2)],
.33   mk-Pck(r,l,a)   → if is-Lbl((c Reg)(l)) ∧ is-LOC((c Reg)(a))
.34                      then (def f: mk-FCT((c Reg)(l),(c Reg)(a));
.35                            Reg := c Reg + [r ↦ f])
.36                      else exit(mk-Lbl(ERROR)),
.37   mk-Upk(l,a,r)   → if is-FCT((c Reg)(r))
.38                      then (def l': s-Lbl((c Reg)(r)),
.39                                d : s-LOC((c Reg)(r));
.40                            Reg   := c Reg + [l ↦ l',a ↦ a'])
.41                      else exit(mk-Lbl(ERROR)),
.42   mk-Pr(rq)       → (def q : is-Reg(rq) → (c Reg)(rq),
.43                              T          → rq;
.44                      Out := c Out^<q>),
      ...

34.0 eval-adr[mk-Adr(b,d)] ≜
  .1   (def il : (c Reg)(b);
  .2    if is-LOC(il) ∨ is-Int(il)
  .3       then (let ea = d+il in
  .4             if -2^s < ea < +2^s
  .5                then return(ea)
  .6                else exit(mk-Lbl(ERROR)))
  .7       else exit(mk-Lbl(ERROR))
```

-- THE COMPILING ALGORITHM

Having now examined the target machine, TM, architecture, that is the semantics of the machine language, independently of SAL, we now turn to the specification of what *Code* to generate for each SAL construct. We

are seeking a definition, *c-prog*, *c-expr*, etc., which again is to be understood in just one, the compiling phase, way. *DICT*ionaries are used as before, and so is *LN*. An extra (compiletime) object is passed to any macro invocation of *c-expr*. It represents the current stack index to the target machine realization of the *TL*s of *DSA*s. Since storage cannot (in general) be reclaimed when a *Block* body *VAL*ue has been computed, and since in this version we have decided to stick with the merge of the control information of the activations *(CP,EP,RA)* not only with local *VAR*iable (*VR*), but also with temporaries (*TL*), we shall have to set aside, in linear storage, the maximum amount of storage cells needed in any expression elaboration, and let that be the over-cautious realization, at this stage, of *TL*. To that end a crude compiler function, *depth*, is defined. It computes the number of temporaries, *de*, needed to compute any expression, but takes into account that embedded *Let*s and *Rec*s lead to new activations for which separate stacks, *TL*, are set aside. We say that *depth* is crude since optimizing versions are easy to formulate, but would, in this example, lead to an excessive number of formulae lines. The disjoint *DSA*s of the previous (IV) definition are now mapped onto a linear ('cell') storage. Each 'new' *DSA* realization consists of 4 + *de* cells: *CP*, *EP*, *RA*, *VR*, respectively *TL*.

VI Compiling Algorithm

VI.1 Compiler Domains

VI.1.1 Syntactic Domains -- as in I.1

VI.1.2 Compiler Components

(1) Σ_C = Ls $\vec{m}$ *Lbl-set*
(2) *DICT* = *Id* $\vec{m}$ *LN*
(3) *LN* = *Nat0*
(4) *Lbl* ∈ *Token* | ERROR

VI.2 Auxiliary Compiler Functions

(5) *dcl* Ls := {ERROR} *type Lbl-set;*

(6) *type: make-lbl:* → $(\Sigma_C \to \Sigma_C \times Lbl)$

(7) *type: depth:* *Expr* → *Nat1*

```
6.0  make-lbl() ≜
       (def l∈Lbl -c Ls;
        Ls := c Ls ∪ {l};
        return(l))

7.0  depth(e) ≜
       cases e:
          mk-Const()        → 1,
          mk-Var()          → 1,
          mk-Infix(e1,e2)   → max(depth(e1),depth(e2)) + 1,
          mk-Cond(t,c,a)    → max(depth(t),depth(c),depth(a)) + 1,
          mk-Let(,d,)       → depth(d),
          mk-Rec(,,)        → 1,
          mk-Lamb(,)        → 1,
          mk-Appl(f,a)      → max(depth(f),depth(a)) + 1
```

VI.3 Translator Specifications

VI.3.1 Global Constant Definitions

```
(8)  let p          = 0,
(9)      ep         = 1,
(10)     ra         = 2,
(11)     vr,pm,u,j  = 3,
(12)     top        = 4,
(13)     br         = 5,
(14)     t          = 4,
(15)     error      = ERROR;
```

VI.3.2 Compiling Specifications

(17) *type: c-prog:* Θ → $(\Sigma_C \to \Sigma_C \times Ins^*)$

(18) *type: c-const:* Θ *STK* → *Ins**

(19-27) *type: c-expr:* Θ *DICT LN STK* → $(\Sigma_C \to \Sigma_C \times Ins^*)$

where Θ stands for the syntactic category name, i.e.: *c-prog* ⊃ Θ =*prog*, *c-expr* ⊃ Θ =*expr*, etc.

(17)

```
c-Prog[e]≜
   (def lexit : make-lbl();
    let de    = depth(e) in
    <mk-Lim(p,0),
     mk-Lim(cp,0),
     mk-Lim(top,t+de)>^
     c-Expr[e]([],0,t)^
    <mk-Ld(u,1,mk-Adr(p,t)),
     mk-Pr(u),
     mk-Jmp(lexit),
     error,
     mk-Br(error),
     lexit>)
```

(18)

```
c-Const[mk-Const(k)](stk)≜
   <mk-Sim(mk-Adr(p,stk),k),
    mk-Lim(u,k)>
```

(20)

```
c-Infix[mk-Infix(e1,o,e2)](δ,ln,stk)≜
   (c-Expr[e2](δ,ln,stk)^
    c-Expr[e1](δ,ln,stk+1)^
   <mk-Ld(u,1,mk-Adr(p,stk+1))
    mk-Fct(u,o,mk-Adr(p,stk)),
    mk-St(mk-Adr(p,stk),u,1)>)
```

(22)

```
c-lamb[mk-Lamb(id,d)](δ,ln,stk)≜
   (def lfct: make-lbl();
        lbyp: make-lbl();
    <mk-Jmp(lbyp),
     lfct>^
     c-Block[d](δ+[id→ln+1],ln+1,stk)
    <mk-Jmp(ra),
     lbyp,
     mk-Lim(u,lfct),
     mk-Pck(u,u,p),
     mk-St(mk-Adr(p,stk),u,1)>)
```

(19)

```
c-Var[mk-Var(id)](δ,ln,stk)≜
   (let n = δ(id) in
    def lloop: make-lbl(),
        lload: make-lbl();
    <mk-Lim(j,ln-n),
     lloop,
     mk-Cjp(j,ZERO,lload),
     mk-Ld(ep,1,mk-Adr(ep,ep)),
     mk-Adj(j,-1),
     mk-Jmp(lloop),
     lload,
     mk-Ld(u,1,mk-Adr(ep,vr)),
     mk-St(mk-Adr(p,stk),u,1),
     mk-Mov(ep,p)>)
```

(21)

```
c-Cond[mk-Cond(t,c,a)](δ,ln,stk)≜
   (def lalt: make-lbl(),
        lout: make-lbl();
     c-Expr[t](δ,ln,stk)^
    <mk-Ld(u,1,mk-Adr(p,stk)),
     mk-Cjp(u,FALSE,lalt)>^
     c-Expr[c](δ,ln,stk)^
    <mk-Jmp(lout),
     lalt>^
     c-Expr[a](δ,ln,stk)^
    <lout>)
```

(23)

```
c-appl[mk-Appl(f,a)](δ,ln,stk)≜
   (def lret: make-lbl();
    c-expr[a](δ,ln,stk)^
    c-expr[f](δ,ln,stk+1)^
    <mk-Ld(u,1,mk-Adr(p,stk+1)),
     mk-Cjp(u,NOTFCT,error),
     mk-Unp(br,ep,u),
     mk-Lim(ra,lret),
     mk-Ld(pm,1,mk-Adr(p,stk)),
     mk-Jmp(br),
     lret>)
```

(24)

```
c-Let[mk-Let(id,d,b)](δ,ln,stk)≜
   (c-Expr[d](δ,ln,stk)^
   <mk-Ld(u,1,mk-Adr(p,stk))>^
    c-Block[b](δ+[id→ln+1],ln+1,stk)
```

(25)

```
c-Rec[mk-Rec(g,lf,b)](δ,ln,stk) ≜
  (let mk-Lamb(id,d) = lf in
   def lfct: make-lbl(),
       lbyp: make-lbl();
   let db = depth(b) in
   <mk-Jmp(lbyp),
    lfct>^
    c-Block[d](δ + [g→ln+1,id→ln+2],
                ln+2,stk)
  ^<mk-Jmp(ra),
    lbyp,
    mk-Lim(u,lfct),
    mk-St(mk-Adr(top,p),p,t-1),
    mk-Pck(u,u,top),
    mk-St(mk-Adr(top,u),u,1),
    mk-Mov(p,top),
    mk-Mov(ep,top),
    mk-Adj(top,t+db)>^
    c-Expr[b](δ + [g→ln+1],ln+1)^
   <mk-Ld(u,1,mk-Adr(p,t)),
    mk-Ld(p,t-1,mk-Adr(p,p)),
    mk-St(mk-Adr(p,stk),u,1)>)
```

(26)

```
c-Block[bl](δ,ln,stk)≜
   (let dbl = depth(bl) in
    <mk-St(mk-Adr(top,p),p,t),
     mk-Mov(p,top),
     mk-Mov(ep,top),
     mk-Adj(top,t+dbl)>^
     c-Expr[bl](δ,ln+1,t)^
    <mk-Ld(u,1,mk-Adr(p,t)),
     mk-Ld(p,t-1,mk-Adr(p,p)),
     mk-St(mk-Adr(p,stk),u,1)>)
```

(27)

```
c-Expr[e](δ,ln,stk)≜
  (is-Const[e] → c-Const[e](stk),
   is-Var[e]   → c-Var[e](δ,ln,stk),
   is-Infix[e] → c-Infix[e](δ,ln,stk),
   is-Cond[e]  → c-Cond[e](δ,ln,stk),
   is-Let[e]   → c-Let[e](δ,ln,stk),
   is-Rec[e]   → c-Rec[e](δ,ln,stk),
   is-Lamb[e]  → c-Lamb[e](δ,ln,stk),
   is-Appl[e]  → c-Appl[e](δ,ln,stk))
```

9.4.2 An Attribute Semantics (VII)

By an attribute semantics definition of a source language is normally understood a set of (usually concrete, BNF) syntax rules defining the source language's character string representations; an association of so-called attributes to each syntactic category (that is distinct rule); and to each pairing of a left-hand side (or: non-terminal) with a right-hand side alternative, a set of action clusters, one per attribute associated with non-terminals of either the leftor the right-hand sides. The action clusters are statement sequences, and their purpose is to assign values to the attributes. The meaning of such an attribute semantics

definition is as follows: consider a source text and its corresponding ('annotated') parse tree. To each tree node allocate an attribute variable corresponding to each of the attributes of the node category. Then compute the values of these according to the attribute semantics definition action clusters. Two extreme cases arise: the value of an attribute is a function solely of the attribute values of the immediate descendant, or ascendant-node(s). We say that the attribute is synthesized, respectively inherited. Obviously nonsensical attribute semantics definitions can be constructed for which their computation for arbitrary or certain parse trees is impossible due for example to circularity. Some such possibilities, for example that of circularity, can, however, be statically checked, that is without recourse to parse trees.

We first choose the same basic realization as up till now, but, for sake of notational variety, and perhaps also your increased reading ability, express the compiled target code in "free form". Hence the meaning is intended to be identical, down to individual computation sequences. The reader will otherwise observe a close resemblance between this, and the immediately preceding definition. In fact their only difference is one of style. Either could equally rightfully be called an attribute semantics.

Annotation:

A concrete, BNF-like grammar is given below. To each category is then associated a small number of attributes. The *depth* attribute, *d*, computes, as did the *depth* function (VI.7), the maximum length of the temporary list - and does so bottom-up; hence it is a synthesized attribute. The *stack*, *level* number and *dictionary* attributes: *stk*, *ln*, *dict* are all passed down from the parse tree root, and are thus inherited. Finally the *code* attribute is synthesized and stores the generated *Codetext* strings. We have not shown a formal (say BNF-) grammar for these strings, but really ought to have done. Subsect. VII.3 finally gives the actual action cluster rules for each grammar rule/production.

Note:

Note also our distinction, in VII.3 formulae between *italic* and roman formulae text parts. The latter denotes *Code-text* to be generated, the former auxiliary quantities whose values are to be resolved in the *code* attribute computation process. Thus in for example VII.8.4 *cde* is to

be computed and its arabic numeral representation then to be inserted. Similarly for lines VII.8.10, 10.5 and 10, where appropriately roman unique label identifiers are to be inserted in lieu of the *italic* label identifiers. The result of a parse tree computation is finally accumulated in *code* of the root node.

VII Synthesized and Inherited Attribute Semantics Compiling Algorithm

VII.1 Concrete BNF-like Grammar

(1) Prog ::= Expr

(2.1) Expr ::= k
.2 ::= id
.3 ::= (Expr + Expr)
.4 ::= if Expr then Expr else Expr fi
.5 ::= let id = Expr ; Block end
.6 ::= rec g = Lamb ; Block end
.7 ::= Lamb
.8 ::= apply Expr (Expr)

(3) Block ::= Expr

(4) Lamb ::= fun (id) = Block end

where we have abbreviated the classes of identifiers, constants and operators to just id, k and +.

VII.2 Node Attributes

(5)	Prog	*code*	*type* *Code-text*	*synthesized*
(6)	Expr	*code*	*type* *Code-text*	*synthesized*
	Lamb	*ln*	*type* *Nat*	*inherited*
		dict	*type* *Id* $\overset{m}{\to}$ *Nat*	*inherited*
		stk	*type* *Nat1*	*inherited*
		d	*type* *Nat1*	*synthesized*
(7)	Block	*code*	*type* *Code-text*	*synthesized*
		ln	*type* *Nat*	*inherited*
		dict	*type* *Id* $\overset{m}{\to}$ *Nat*	*inherited*

VII.3 Action Cluster Rules

```
(8)   Progp ::= Expre
  .1  def lexit: make-lbl();
  .2  code_p := "R[p] := 0;
  .3            R[ep] := 0;
  .4            R[top] := t + cde;
  .5           "^ccode_e^"
  .6            R[u] := cS[cR[p] + t];
  .7            Out := cR[u];
  .8            goto lexit;
  .9            lerr:
  .10           Out := ERROR;
  .11           lexit:
  .12          ";
  .13 ln_e   := 0;
  .14 dict_e := [];
  .15 stk_e  := t;

(9)   Expre ::= k
  .1  de     := 1;
  .2  code_e := "S[cR[p]+ cstk_e] := k;
  .3            R[u] := k;
  .4           ";

(10)  Expre ::= id
  .1  def lloop: make-lbl(),
  .2      lload: make-lbl();
  .3  de     := 1;
  .4  code_e := R[j]              := cln_e - cdict_e(id);
  .5            lloop:
  .6            if cR[j] = 0 then goto lload;
  .7            R[ep]             := cS[cR[ep] + ep];
  .8            R[j]              := cR[j] - 1;
  .9            goto lloop;
  .10           lload:
  .11           R[u]              := cS[cR[ep] + vr];
  .12           S[cR[p] + cstk_e] := cR[u];
  .13           R[ep]             := cR[p];
  .14          ";
```

```
(11)   Expre ::= ( Expre1 + Expre2 )
 .1    de              := max(cde^1, cde^2) + 1;
 .2    ln_e1, ln_e2    := cln_e;
 .3    stk_e2          := cstk_e;
 .4    stk_e1          := cstk_e + 1;
 .5    dict_e1, dict_e2 := cdict_e;
 .6    code_e := ccode_e2 ^
 .7              ccode_e1 ^
 .8              "R[u]                := cS[cR[p] + cstk_e];
 .9               R[u]                := cR[u] + cS[cR[p] + cstk_e];
 .10              S[cR[p] + cstk_e]   := cR[u];
 .11             ";

(12)   Expre ::= if Exprt then Exprc else Expra
 .1    def lalt: make-lbl(),
 .2        lout: make-lbl();
 .3    de                    := max(cd_t, cd_c, cd_a) + 1;
 .4    ln_t, ln_c, ln_a      := cln_e;
 .5    stk_t, stk_c, stk_a   := cstk_e;
 .6    dict_t, dict_c, dict_a := cdict_e;
 .7    code_e := ccode_t ^
 .8              "R[u] := cS[cR[p] + cstk_e];
 .9               if ¬cR[u] then goto lalt;
 .10           "^code_c ^"
 .11-.14          goto lout; lalt: "^ccode_a ^" lout:
 .15           ";

(13)   Expre ::= let id = Exprd ; Blockb end
 .1    d_e     := cd_b;
 .2    ln_a    := cln_e;
 .3    ln_b    := cln_e + 1;
 .4    stk_d   := cstk_e;
 .5    dict_d  := cdict_e;
 .6    dict_b  := cdict_e + [id ↦ cln_e + 1];
 .7    code_e  := ccode_d ^"
 .8               R[u] := cS[cR[p] + cstk_e];
 .9             "^ccode_b^"
 .10             ";
```

```
(14)  Expre ::= rec g = fun ( id ) = Blockd end ; Blockb end
 .1   def lfct: make-lbl(),
 .2       lbyp: make-lbl();
 .3   de    := 1;
 .4   lnd   := cln_e + 2;
 .5   lnb   := cln_e + 1;
 .6   dictd := cdict_e + [g ↦ ln+1, id ↦ ln+2];
 .7   dictb := cdict_e + [g ↦ ln+1];
 .8   code_e := "goto lbyp;
 .9              lfct:
 .10           "^ccoded^"
 .11             goto cR[ra];
 .12             lbyp:
 .13             R[u]              := lfct;
 .14             R[u]              := mk-FCT(cR[u],cR[top]);
 .15             S[cR[top] + p]    := cR[p];
 .16             S[cR[top] + ep]   := cR[ep];
 .17             S[cR[top] + ra]   := cR[ra];
 .18             S[cR[top] + vr]   := cR[u];
 .19             R[p]              := cR[top];
 .20             R[ep]             := cR[top];
 .21             R[top]            := cR[top] + (t+cd_e);
 .22           "^ccodeb^"
 .23             R[ep]             := cS[cR[p] + ep];
 .24             R[ra]             := cS[cR[p] + ra];
 .25             R[u]              := cS[cR[p] + t];
 .26             R[p]              := cS[cR[p] + p];
 .27             S[cR[p] + cstk_e] := cR[u];
 .28           ";

(16)  Expre ::= Lambl
 .1   de    := cdl;
 .2   lnl   := cln_e;
 .3   stkl  := cstk_e;
 .4   dictl := cdict_e;
 .5   code_e := ccodel;
```

```
(15)  Lambe ::= fun ( id ) = Blockb  end
 .1   def lfct : make-lbl(),
 .2        lbyp : make-lbl();
 .3   de    := 1;
 .4   lnb   := clne + 1;
 .5   dictb := cdicte + [id ↦ clne + 1];
 .6   codee := "goto lbyp;
 .7            lfct:
 .8           "^ccodeb^"
 .9            goto cR[ra];
 .10           lbyp:
 .11           R[u]                  := lfct;
 .12           R[u]                  := mk-FCT(cR[u],cR[p]);
 .13           S[cR[p] + cstke] := cR[u];
 .14          ";

(17)  Expre ::= apply  Exprf  ( Expra )
 .1   def lret : make-lbl();
 .2   de           := max(cdf,cda) + 1;
 .3   lnf,lna     := clne;
 .4   stka         := cstke;
 .5   stkf         := cstke + 1;
 .6   dictf,dicta := cdicte;
 .7   codee        := ccodea^"
 .8                  "^ccodef^"
 .9                    R[u]  := cS[cR[p] + (cstke+1)];
 .10                   IF NOTFCT(cR[u]) THEN GOTO lerr;
 .11                   R[br] := s-Lbl(cR[u]);
 .12                   R[ep] := s-LOC(cR[u]);
 .13                   R[ra] := lret;
 .14                   R[pm] := cS[cR[p] + cstke];
 .15                   GOTO cR[br];
 .16                   lret:
 .17                 ";

(18)  Blockb ::= Expre
 .1   lne   := clnb + 1;
 .2   dicte := cdictb;
 .3   stke  := t;
 .4   codeb := "S[cR[top] + p]   := cR[p];
 .5            S[cR[top] + ep]  := cR[ep];
```

```
.6          S[cR[top] + ra]  := cR[ra];
.7          S[cR[top] + vr]  := cR[u];
.8          R[p]             := cK[top];
.9          R[ep]            := cR[p];
.10         R[top]           := cR[top] + (t+cd_e);
.11       "^ccode_e^"
.12         R[e]P            := cS[cR[p] + ep];
.13         R[ra]            := cS[cR[p] + ra];
.14         R[u]             := cS[cR[p] + t];
.15         R[p]             := cS[cR[p] + p];
.16         S[cR[p] + cstk_e] := cR[u];
.17       ";
```

9.4.3 Another Attribute Semantics (VIII)

The language defined by the concrete BNF grammar given in 9.4.2 (VII.1) for SAL is both bottom-up and topdown analyzable. That didn't matter very much in section 9.4.2, since attribute variable value computations still required the presence of the entire parse tree before any *Code-text* could be generated. In this section we present an attribute semantics specification of another compiling algorithm, which, based on a top-down parse process, is capable of generating *Code-text* simultaneously with parsing. Again we shall not argue how we choose a/the solution. Instead we ask you to recall the twin stack abstract machine of section 9.3.3. Now all *DSA* realizations fit exactly into four (*t*) positions: *(CP, EP, RA, VR)* with temporaries allocated to a global, contiguous stack, *STK*'s direct implementation. Since SAL is simply applicative (it permits for example no GOTOs) this poses no special problems as concerns correct indices into stack tops. The *STK* has been realized in 'core' "below" the activation stack: think of the target machine addressing being "wrapped around" zero address to maximum available core storage address - and you get a scheme which was very common in the earlier days on mono-processing. One crucial, final note: to cope with known *Code-text* to be "delay"-generated a global 'attribute' (also) called *code*, is introduced. It is treated as a stack. Push corresponds to concatenation, pop to taking the head off -- leaving the tail. Pushing occurs for all *Code-texts* known when recognizing the initial prefix string, as one does in top-down analysis, of a composite expressing: if, let, rec, apply, (and fun. Popping of one part occurs when any expression has been completely analyzed: k, id, fi, end, end, end,), and end, respectively.

VIII Inherited Attribute Semantics Compiling Algorithm

VIII.1 Syntactic Domains -- as in VII.1

VIII.2 Node & Global Attributes

(5) Expr *ln* *type Nat0* *inherited*
Lamb *dict* *type Id* $\overrightarrow{m}$ *Nat0* *inherited*
Block

(6) Prog *code* *type Code-text* *stack*
print *type Code-text* *output*

VIII.3 Attribute Rules

(7) Progp ::= Expre
.1 *def lexit: make-lbl();*
.2 *print* "R[p] := 0;
.3 R[ep] := 0;
.4 R[top] := *t*;
.5 R[stk] := -1;";
.6 *code* := <"R[u] := cS[-1];
.7 Out := cR[u];
.8 goto *lexit;*
.9 *lerr:*
.10 Out := ERROR;
.11 *lexit:* ">;
.12 ln_e := *0;*
.13 $dict_e$:= [];

(8) Expre ::= k
.1 *print* "S[cR[stk]] := k;
.2 R[u] := k;
.3 R[stk] := cR[stk] + 1;
.4 "^*hd c code;*
.5 *code* := *tl c code;*

(9) Expr ::= id
.1 *def lloop* : *make-lbl(),*
.2 *lload* : *make-lbl();*
.3 *print* "R[j] := cln_e - ($cdict_e$)(id);

```
.4          lloop:
.5          if cR[j] = 0 then goto lload;
.6          R[ep] := cS[cR[ep] + ep];
.7          R[j]  := cR[j] - 1;
.8          goto lloop
.9          lload:
.10         R[u]  := cS[cR[ep] + vr];
.11         S[cR[stk]] := cR[u];
.12         R[stk] := cR[stk] - 1;
.13         R[ep]  := cR[p];
.14       "^hd c code;
.15 code := tl c code;
```

```
(10)  Expre ::= ( Expr1 + Expr2 )
.1    ln_e1, ln_e2     := cln_e;
.2    dict_e1, dict_e2 := cdict_e;
.3    code             := <"">^
.4                        <"R[u]        := cS[cR[stk]];
.5                         R[u]        := cR[u] + cS[cR[stk] + 1];
.6                         R[stk]      := cR[stkP + 1;
.7                         S[cR[stk]] := cR[u];
.8                        ">^ccode;
```

```
(11)  Expre ::= if Exprt then Exprc else Expra fi
.1    def lalt : make-lbl(),
.2        lout : make-lbl();
.3    ln_t, ln_c, ln_a       := cln_e;
.4    dict_t, dict_c, dict_a := cdict_e;
.5    code                   := "R[u]   := cS[cR[stk]];
.6                               R[stk] := cR[stk] + 1;
.7                               if ¬cR[u] then goto lalt;
.8                              ">^
.9                             <"goto lout;
.10                             lalt:
.11                            ">^
.12                            <"lout:
.13                             ">^ccode;
```

```
(12)   Expre ::= let id = Exprd ; Blockb end
 .1    ln_d   := c-ln_e;
 .2    ln_b   := c-ln_e + 1;
 .3    dict_d := c-dict_e;
 .4    dict_b := c-dict_e + [id ↦ c-ln_e + 1];
 .5    code   := <"R[u]   := cS[cR[stk]];
 .6               R[stk] := cR[stk] + 1;
 .7              ">^
 .8             <"">^c-code;

(13)   Expre ::= rec g =  fun ( id ) = Blockd end ; Blockb end
 .1    def lfct : make-lbl(),
 .2        lbyp : make-lbl();
 .3    ln_d   := c-ln_e + 2;
 .4    ln_b   := c-ln_e + 1;
 .5    dict_d := c-dict_e + [id ↦ c-ln_e + 2, g ↦ c-ln_e + 1];
 .6    dict_b := c-dict_e + [g  ↦ c-ln_e + 1];
 .7    print "goto lbyp;
 .8                lfct:
 .9          ";
 .10   code := <"goto cR[ra];
 .11             lbyp:
 .12             R[u]              := lfct;
 .13             R[u]              := mk-FCT(cR[u],cR[top]);
 .14             S[cR[top] + p]    := cR[p];
 .15             S[cR[top] + ep]   := cR[ep];
 .16             S[cR[top] + ra]   := cR[ra];
 .17             S[cR[topP + vr]   := cR[u];
 .18             R[p]              := cR[top];
 .19             R[ep]             := cR[top];
 .20             R[top]            := cR[top] + t;
 .21           ">^<"
 .22             R[ep]             := cS[cR[p] + ep];
 .23             R[ra]             := cS[cR[p] + ra];
 .24             R[u]              := cS[cR[p] + t];
 .25             R[p]              := cS[cR[p] + p];
 .26             S[cR[stk]]        := cR[u];
 .27             R[stk]            := cR[stk] + 1;
 .28           ">^c-code;
```

```
(14)  Expre ::= Lambd
  .1  ln_d   := c̲ln_e;
  .2  dict_d := c̲dict_e;

(15)  Lambe ::= fun ( id ) = Blockb end
  .1  def lfct : make-lbl(),
  .2      lbyp : make-lbl();
  .3  ln_b   := c̲ln_e + 1;
  .4  dict_b := c̲dict_e + [id ↦ c̲ln_e + 1];
  .5  code   := "goto c̲R[ra];
  .6            lbyp:
  .7            R[u]        := lfct;
  .8            R[u]        := mk-FCT(c̲R[u],c̲R[p]);
  .9            S[cR[stk]]] := c̲R]u];
  .10           R[stk]      := c̲R[stk] - 1;
  .11         ">^c̲code;
  .12 print "goto lbyp;
  .13        lfct:
  .14       ";

(16)  Expre ::= apply Exprf ( Expra )
  .1  def lret : make-lbl();
  .2  ln_f,ln_a     := c̲ln_e;
  .3  dict_f,dict_a := c̲dict_e;
  .4  code          := <"">^
  .5                   <"R[u] := c̲S[c̲R[stk] - 1];
  .6                     if NOTFCT(c̲R[u])
  .7                        then goto lerr;
  .8                     R[br]  := s-Lbl(c̲R[u]);
  .9                     R[ep]  := s-LOC(c̲R[u]);
  .10                    R[ra]  := lret;
  .11                    R[pm]  := c̲S[c̲R[stk]];
  .12                    R[stk] := c̲R[stk] - 2;
  .13                    goto c̲R[br];
  .14                    lret:
  .15                  ">^c̲code;
```

```
(17)   Blockb ::= Expre
  .1   lne    := clnb + 1;
  .2   dicte  := cdictb;
  .3   print "S[cR[top] + p]  := cR[p];
  .4          S[cR[top] + ep] := cR[ep];
  .5          S[cR[top] + ra] := cR[ra];
  .6          S[cR[top] + vr] := cR[u];
  .7          R[p]            := cR[top];
  .8          R[ep]           := cR[top];
  .9          R[top]          := cR[top] + t;
  .10       ";
  .11  code := <"R[ep]       := cS[cR[p] + ep];
  .12           R[ra]       := cS[cR[p] + ra];
  .13           R]u]        := cS[cR[p] + t];
  .14           R[p]        := cS[cR[p] + p];
  .15           S[cR[stk]] := cR[u];
  .16           R[stk]      := cR[stk] + 1;
  .17         ">^ccode;
```

9.5 COMPILER STRUCTURES

From the compiling algorithm specifications of section 9.3.2-3 we can now read properties other than just the source text input vs. target machine code output itself. Thus the compiling algorithm determines first level structures of the compiler itself. That is, it answers questions such as: "Is it a two, or can it be a single-pass compiler?"; "What information is put in the dictionary, and when?"; "How is the dictionary realizable: as part of the intermediate text of a multi-pass compiler, with the *dict*ionary components 'scattered' over this intermediate text; or necessarily as a 'global' component, 'disjoint' from any intermediate text or parse tree?".

Single- and Multi-Pass Compilers

If all attributes can be computed in a synthetic manner then a single-pass compiler based on a bottom-up parse can always be realized. This is so since any deterministic language can always be so (in fact LR(K)) parsed. Similarly, if the language can be top-down (for example LL(K)) parsed (possibly using some recursive descent method), and a compiling algorithm given by a purely inherited attribute semantics, then a single-

pass compiler is again possible. If, however, as for example suggested by our first SAL attribute semantics compiling algorithm (9.3.2), some attributes, like the local, temporay list stack pointer, and the block/body level number, and *dict*ionary, are inherited, while others, the maximum local stack depth and the generated *code*, are synthesized, then a multi-pass compiler with at least two passes cannot be avoided. If the inherited and synthesized attributes of such a compiling algorithm solely derive from constants emanating from respectively the root and leaves, then a two-pass compiler can result. This is in fact the case with the VI and VII specifications. The exact minimum number of logical, that is intrinsically required, passes, is a function of the semantics and syntax of the language - with the semantics property eventually showing up in the intricate web-like relationships between synthesized and inherited attributes. Of course: silly, unnecessarily complicated, attribute semantics, that is such as those involving nonintrinsical combinations of inherited and synthesized attribute value computations, would then indicate a higher (minimum) number of passes than strictly required. Only calm scrutiny, a careful analysis and a complete mastering of the language semantics and specification tools will eventually lead to optimal realizations.

So what is then the difference between the two compiling algorithms: VII and VIII? One leads to a two-pass-, the other to a single-pass compiler for one and the same language (semantics). Is not a minimum pass compiler always to be preferred? Well, the gain in compilation speed in the latter has been obtained at the expense of slower execution speed, since -- as in usually the case with pure, stack-oriented execution -- a temporary stack index, R[stk], must now be dynamically adjusted: at the worst once per 'popping', and once per 'pushing'. This being in contrast to the fixed-offset addressing possible by our two-pass compilation, which - in turn - causes possibly excessive storage to be (pre-)allocated.

Compiler Object Realizations

Whilst the purpose of 'mapping' the denotational semantics into successively lower-order, increasingly more concrete/intricate semantics definitions, was one of eliminating higher-order, functional objects -- as well as the run-time presence of elaboration routines -- we now suddenly see the re-appearance, in for example the attribute semantics definitions, of higher order objects: the functional dictionaries, and even the (non-functional, but) varying string length (*Code-text*) obobjects. The input/

output relationships of a compiler have been specified. Now we must object transform abstractly described compiler objects, and operation decompose the likewise implicitly specified primitive operations on these. In fact, we must also take issue, at long last, with the internal realization of abstract SAL programs. But note this: nothing has been lost in postponing this decision till now. On the contrary: we may now be able to design exactly that internal representation which best suits the code-generation parse-tree walking algorithm. The techniques of 'mapping' such abstractions into concrete implementations using methods akin to those of this papers' specialized, run-time structure-oriented ones, will not be further dealt with here.

9.6 COMPILING CORRECTNESS

Time has finally come to take issue with the problem of correctness. In this section we shall illustrate only one such proof. We prove that the development I → II is correct. Subsequent development stages are proved correct using essentially the same technique, but becoming, first increasingly more cumbersome (to report and read), and with III → IV also somewhat more complex.

Correctness Criterion:

-- Theorem (Thm)

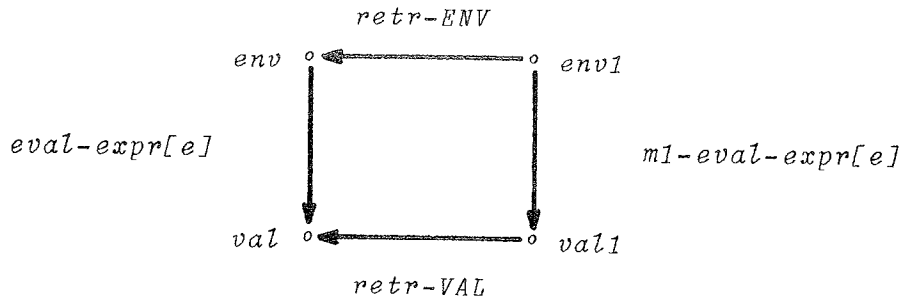
$$
\begin{array}{l}
(\forall env \in ENV,\ \forall env1 \in ENV1) \\
\quad (env = retr\text{-}ENV(env1)) \\
\qquad \supset\ (\forall e \in expr)(eval\text{-}expr[e]env \\
\qquad\quad = retr\text{-}VAL(m1\text{-}eval\text{-}expr[e](env1)))
\end{array}
$$

Annotation:

For all such abstract, *env*, and concrete, *env1*, environments which correspond, it shall be the case that evaluating any expression, *e*, using the abstract interpreter on *env* will yield the same value as is retrievable from evaluating that same expression *e* using the more concrete interpreter on *env1*.

The statement of this criterion, as well as its actual, detailed proof, is new.

The structure of Thm can be pictured:



It is derived from the general idea of 'simulation' of one program by another (executing one algebra on another).

Proof of Correctness

Our task of proving that *m1-eval-expr* (in the sense of the criterion above) does the same, to any expression, as *eval-expr*, that is delivers comparable results, can be broken into two steps. Firstly we plan the proof. We look for a strategy, a proof technique, which, with the least amount of effort, will achieve the proof. Secondly we carry through the actual details of all parts of the proof, as structured in the first step above. The strategy to be followed here is that of proof by structural induction. That is: since the criterion calls for all expressions, we naturally look for a way of proving it for a finite set of representative examples. The selection of these is guided by the structure (and alternatives) of the syntactic domain abstract syntax, and by the corresponding structure and alternatives of the elaboration functions. The structure of the semantic domains here play a lesser rôle (than might otherwise be the case -- for for example III → IV). The abstract syntax for expressions determined the major structure of the elaboration functions, as is natural for denotational semantics definitions. Therefore we shall now structure the proof according to the cases of alternative syntactic categories, and by induction due to the recursive definition of *Expr*.

Legend:

Thm., abbreviates 'theorem', Asm. 'assumption', Lm. 'lemma', QED 'proved'; the lemma is displayed after the proof.

case 1: *e = mk-Const(k)*

1. *eval-expr[mk-Const(k)](env)* = *k*
2. *m1-eval-expr[mk-Const(k)](env1)* = *k*
3. *retr-VAL(k)* = *k*

case 2: *e = mk-Var(id)*

0. assumption: *env = retr-ENV(env1)* Thm.
1. *eval-expr[mk-Var(id)](env)* = *env(id)*
2. *m1-eval-expr[mk-Var(id)](env1)* = *look-up(id,env1)*
3. *retr-VAL(look-up1(id,env1))* = *env(id)* Lm., QED.

case 4: *e = mk-Cond(t,c,a)*

0. assumption: *env = retr-ENV(env1)* Thm.
1. *eval-expr[mk-Cond(t,c,a)](env)* = *if eval-expr[t](env) then eval-expr[c](env) else eval-expr[a](env)*
2. *m1-eval-expr[mk-Cond(t,c,a)](env1)* = *if m1-eval-expr[t](env1) then m1-eval-expr[c](env1) else m1-eval-expr[a](env1)*
3. induction hypothesis Thm.
   .1 *eval-expr[t](env) ~ m1-eval-expr[t](env1)*
   .2 *eval-expr[c](env) ~ m1-eval-expr[c](env1)*
   .3 *eval-expr[a](env) ~ m1-eval-expr[a](env1)*
4. *m1-eval-expr[t](env1) ∈ Bool*
5. *eval-expr[t](env) = m1-eval-expr[t](env1)*
6. subcase 1
   .0 *eval-expr[t](env)* = *true* Asm.
   .1 *eval-expr[mk-Cond(t,c,a)](env)* = *eval-expr[c](env)*
   .2 *m1-eval-expr[mk-Cond(t,c,a)](env1)* = *m1-eval-expr[c](env1)*
   .3 8.1 = 8.2 follows from 5.2 QED.
7. subcase 2 -- as 6. QED.

case 5: *e = mk-Lamb(id,d)*

0. assumption: *env = retr-ENV(env1)* Thm.

1. *eval-expr[mk-Lamb(id,d)]env = eval-fun[mk-Lamb(id,d)]env*
2. *m1-eval-expr[mk-Lamb(id,d)](env1) = mk-CLOS(mk-Lamb(id,d),env1)*
3. *retr-VAL(mk-CLOS(mk-Lamb(id,d),env1)*
 .1 *= eval-fun[mk-Lamb(id,d)](retr-ENV(env1))*
 .2 *= eval-fun[mk-Lamb(id,d)]env*

QED follows from 2. = 5.2

case 6: *e = mk-Appl(f,a)*

0. assumption: *env = retr-ENV(env1)*
1. *eval-expr[mk-Appl(f,a)]env = (eval-expr[f]env)(eval-expr[a]env)*
2.0 *m1-eval-expr[mk-Appl(f,a)](env1)*
 .1 *= (let clos = m1-eval-expr[f](env1),*
 .2 *arg = m1-eval-expr[a](env1) in*
 .3 *apply1(clos,arg))*

3. induction,hypothesis Thm.
 .1 *eval-expr[f]env ~ m1-eval-expr[f](env1)*
 .2 *eval-expr[a]env ~ m1-eval-expr[a](env1)*

4.0 *eval-expr[f]env ∈ FUN ⊃* Asm.
 .1 *(∃mk-Lamb(id,d),∃env' ∈ ENV)*
 .2 *(eval-expr[f]env = eval-fun[mk-Lamb(id,d)](env)*

The above should be expressed in terms of a "recursively defined predicate", see [Milne 77a, Tennent 82a]. The idea of that predicate transpires, however from the above & below predicates.

-- Considering only a "good" case:

5.0 *m1-eval-expr[f](env1) ∈ CLOS ⊃*
 .1 *(...,∃env1' ∈ ENV1)*
 .2 *(env' = retr-ENV(env1'))*
 .3 *∧(m1-eval-expr[mk-Lamb(id,d)](env1') = m1-eval-expr[f](env1)*

6. *m1-eval-expr[mk-Lamb(id,d)](env1')*
 .1 *= mk-CLOS(mk-Lamb(id,d),env1')*
 .2 *= m1-eval-expr[f](env1)*
 .3 *= clos*

7. *apply1(clos,arg)*
 .1 = *apply1(mk-CLOS(mk-Lamb(id,d),env1'),arg)*
 .2 = (*let env1" = <mk-SIMP(id,arg)>^env1'*
 m1-eval-expr[d](env1"))

8. *(eval-expr[f]env)(eval-expr[a]env)*
 .1 = *eval-fun[mk-Lamb(id,d)](env')(eval-...|*
 .2 = *λx.(let env" = env' + [id ↦ a] in |..-expr[a]env)*
 eval-expr[d]env")(eval-expr[a]env)
 .3 = (*let env" = env' + [id ↦ eval-expr[a]env];*
 eval-expr[d]env")

9. *env" = retr-ENV(env1")* follows from:
 .1 *retr-ENV(env1') = env'*
 .2 ∧ *env1" = <mk-SIMP(id,arg)>^env1'*
 .3 ∧ *env" = env' + [id ↦ eval-expr[a]env]*
 .4 ∧ *arg = m1-eval-expr[a](env1)*
 .5 ∧ *retr-ENV* definition
 .6 ∧ *eval-expr[a]env = retr-VAL(arg)*

10. induction, hypothesis:
 .1 *eval-expr[d]env" ~ m1-eval-expr[d](env1")* Thm.

11. QED then follows

case 7: *e = mk-Rec(g,mk-Lamb(id,d),b)*

0. assumption: *env = retr-ENV(env1)*
1. *eval-expr[mk-Rec(g,mk-Lamb(id,d),b)]env*
2. = (*let env' = env + [g ↦ eval-fun[mk-Lamb(id,d)]env] in*
 eval-expr[b]env')
3. *m1-eval-expr[mk-Rec(g,mk-Lamb(id,d),b)](env1)*
4. = (*let env1' = <mk-REC(g,mk-Lamb(id,d))>^env1 in*
 m1-eval-expr[b](env1'))
5. *env1' = retr-ENV(env')* follows from:
 .1 *env' = env + [g ↦ eval-fun[mk-Lamb(id,d)]env']*
 .2 ∧ *env1' = <mk-REC(g,mk-Lamb(id,d))>^env1*
 .3 ∧ *env = retr-ENV(env1)*
 .4 ∧ *retr-ENV(env1')*
6. (5) ⊃ *eval-expr[b]env'*
 = *retr-VAL(m1-eval-expr[b](env1'))* QED.

Lemma:

$(env=retr\text{-}ENV(env1)) \supset (\forall id \in dom\ env)(env(id)=retr\text{-}VAL(look\text{-}up1(id,env1)))$

The lemma is (for example) proved by induction on the length of *env1*. Two cases form the basis step: the *SIMP*le and the *REC*ursive header. We leave the proof as an exercise.

9.7 SUMMARY

We have shown the systematic derivation, from a denotational semantics definition, of an implementation of a Simple Applicative Language featuring both a block-structure and the procedure concept. The derivation proceeded in a number of increasingly concrete, more detailed, run-time-oriented styles, Hence we were able to illustrate how run-time structures such as DISPLAYs could be orderly developed, eventually proven correct. The various definition styles were basically those current during the 1960's, and we have thus shown how they relate.

9.8 BIBLIOGRAPHY

Chapter 1 surveys the roots of denotational semantics definitions. Our example language, SAL, is taken from [Reynolds 72a], as is the next stage of development.

The first first-order applicative semantics was that of LISP1.5 [McCarthy 62b]. (A denotational semantics study of LISP 1.5 was carried out by Gordon [Gordon 73a].) The 1960's saw further exercises in first-order functional semantics, notably among which we find the IBM Vienna Lab. series of PL/I definitions: ULD versions 1,2,3: [ULD66, ULD68, ULD69], Reynolds' GEDANKEN [Reynolds 70a], and the sketches of Lockwood Morris [Morris 70a]. Common to all, however, is the fact that none were derived from other semantic definitions (except perhaps in an intuitive sense those of [Morris 70a]); but marked the only available 'abstraction'. The present derivation of SAL.I into SAL.II is essentially that of [Reynolds 72a], the statement of the retrieve functions and the (proof of the) theorem (sect. 9.6) is ,however, new.

Abstract state machine semantics definitions were first reported by

Landin [Landin 64a], and received their full development with the IBM Vienna Lab. series of PL/I definitions, ULD versions 1,2,3 (Universial Lanuage Descriptions, as expressed in the so-called VDL, Vienna Definition Language [ULD66, ULD687, ULD69]. Whereas Landin's definition of an even simpler applicative expression language, AE, than SAL was also paired with a denotational definition, no attempt was then reported on proving their 'equivalence', let alone deriving the former systematically from the latter. Landin's mechanical version was since referred to as the SECD, Stack/Environment/Control/Dump, machine specification style, since the structure was amenable to a variety of language definitions. The VDL based definitions [ULD66, ULD68, ULD69, Lauer 68a, Zimmerman 69a, Allen 72a] too, were free-standing, in that no abstract model was used as a departure point. The recent PL/I ANS/ECMA standards proposal [ANSI 76a] is basically a derivative of the ULD/VDL style of semantics, as also explained in [Beech 73a]. [Lauer 73a - Hoare 74a, Marcotty 76a] presents examples of abstract state machine semantics, with [Lauer 73a - Hoare 74a] proving equivalence among several variants of these and also axiomatically stated versions.

Historically attribute semantics originated with E.T. Irons' work on ALGOL 60 compilers - and we still find that the technique is mostly used in compiling algorithm specification, including statically checkable context condition/constraint testing. In contrast hereto: to specify the semantics of a source language one usually introduces a parse tree 'walking' function which in addition to the local attributes, also work on, that is manipulate, global objects, thus effecting desired computations.

Attribute semantics definitions received their purifying, individualizing treatment from Knuth, Lewis/Stearns/Rosenkrantz, Wirth/Weber and others; Irons started the whole thing [Irons 61a, Irons 63a, Wirth 66b, Knuth 68a, Knuth 71a, Lorho 75a, Neel 74a, Bochmann 76a, Wilner 72a, Kennedy 74a].

The idea of proving correctness "by commuting diagrams", that is by simulating one machine by another was reported by [McCarthy 67a, Landin 72a, Milner 71a-b, Weyhrauch 72a, Morris 73a, Lucas 72a, Goguen 75a, Goguen 78a], see also [Jones 80c, Ganzinger 80a, Gaudel 80a], and many others. The subject of compiler correctness proving is currently under intense study.

The idea of deriving target machine codes from source language specifications is reported in [Dommergaard 80a, Wand 80bc, Wand 82b, Mosses 81a].