Compositional Compiler Construction: Oberon0

Marcos Viera

S. Doaitse Swierstra

Technical Report UU-CS-2012-016
October 2012

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

Compositional Compiler Construction: Oberon0

Marcos Viera

Instituto de Computación
Universidad de la República
Montevideo, Uruguay
mviera@fing.edu.uy

S. Doaitse Swierstra

Department of Computer Science
Utrecht University
Utrecht, The Netherlands
doaitse@cs.uu.nl

Abstract

We describe an implementation of an Oberon0 compiler using the techniques proposed in the CoCoCo project. The compiler is constructed out of a collection of pre-compiled, statically type-checked language-definition fragments written in Haskell.

1 Introduction

As a case study of the techinques proposed in the CoCoCo project[1], we participated in the LDTA 2011 Tool Challenge[2]. The challenge was to implement a compiler for Oberon0, a small (Pascal-like) imperative language designed by Nicolas Wirth as an example language for his book "Compiler Construction" [**?**].

The goal of the challenge is to contribute to "a better understanding, among tool developers and tool users, of relative strengths and weaknesses of different language processing tools, techniques, and formalisms". The challenge is divided into a set of incremental sub-problems, that can be seen as points in a two dimensional space. The first dimension (Table 1) defines a series of language levels, each building on the previous one by adding some new features. The second dimension

L1	Oberon0 without procedures and with only primitive types.
L2	Add a Pascal-style for-loop and a Pascal-style case statement.
L3	Add Oberon0 Procedures.
L4	Add Oberon0 Arrays and Records.

Table 1: Language Levels.

(Table 2) consists of several traditional language processing tasks, such as parsing, pretty-printing, static analysis, optimizations and code generation.

This incremental design has two main reasons. First, participants were able to provide partial solutions, choosing the most suitable tasks to show the characteristics and features of their tool

[1] `http://www.cs.uu.nl/wiki/Center/CoCoCo`
[2] `http://ldta.info/tool.html`

T1	Parsing and Pretty-Printing
T2	Name Binding
T3	Type Checking
T4	Desugaring
T5	C Code Generation

Table 2: Processing Tasks.

or technique. The possible software artifacts generated to solve any of the 25 proposed problems range between a L1 T1, a parser and pretty-printer of a simple subset of Oberon0, and L4 T1-5, the full proposed system. In order to be able to compare participant's artifacts, a list of suggested software artifacts to be completed (Table 3) is provided. The second reason of the design is to

Artifact	*Level*	*Tasks*	*Description*
A1	L2	T1-2	Core language with pretty-printing and name binding
A2a	L3	T1-2	A1 plus pretty-printing and name binding for procedures
A2b	L2	T1-3	A1 plus type checking
A3	L3	T1-3	A2a and A2b
A4	L4	T1-5	Full language and all tasks

Table 3: Artifacts.

show how the different techniques for modularity provided by the participants can be used in the implementation of a growing system.

We have provided an implemention of all the proposed problems, and made it available in Hackage as the `oberon0`[3] package.

2 Architecture

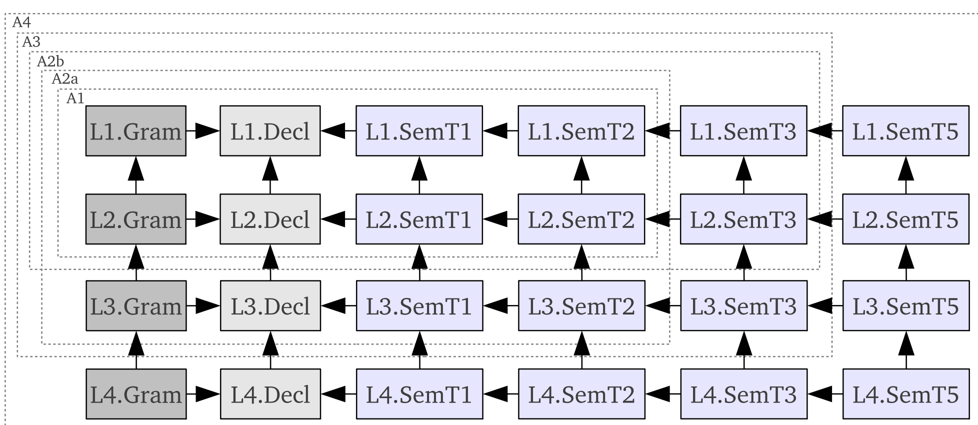

Figure 1: Architecture of the Oberon0 Implementation

The architecture of our implementation of Oberon0 is given in Figure 1; boxes represent Haskell modules and arrows are **import** relations[4], where every module can be compiled separately and

[3] `http://hackage.haskell.org/package/oberon0`.

[4] For example, module *L2.SemT1* imports from (i.e. depends on) modules *L2.Decl* and *L1.SemT1*.

results in a set of normal Haskell value definitions. The design is incremental: rows corresponds to syntactic extensions (language levels) and columns corresponds to semantic extensions (tasks); each artifact in the challenge corresponds to a dashed box surrounding the modules involved in it. For each language level *L1* to *L4*:

- *Gram* modules contain syntax definition in the form of first-class grammar fragments, as introduced in [**?**]
- *Decl* modules contain the definition of the type of the semantics' record, and thus the interface to the corresponding part of the abstract syntax of the language at hand
- *Sem* modules implement the semantics of each task in the form of rules which together construct an attribute grammar, as introduced in [**?**].

Notice that we do not include modules to implement Task 4. In Subsection 4.3 we will explain how by using attribute grammar macros when defining *L2* we get this task almost for free.

To build a compiler, e.g. Artifact 4 (Figure 2), we import the syntax fragments (*l1*, *l2*, *l3* and *l4* from *L4.Gram*) and their respective semantics (*l1t4*, *l2t4*, *l3t4* and *l4t4* from *L4.Sem*), combine them and build the compiler in the form of a parser which calls semantic functions. In Figure 3

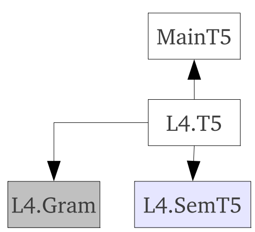

Figure 2: Architecture of Artifact 4

```
gl4t5 = closeGram $ emptyGram +>>
                    l1 l1t5   +>>
                    l2 l2t5   +>>
                    l3 l3t5   +>>
                    l4 l4t5
pA4 = (parse . generate kws) gl4t5
```

Figure 3: A Parser for Artifact 4

we show how the parser of Artifact 4 is generated. The left-associative operator (`+>>`) composes an initial grammar with an extension; we start with an empty grammar (*emptyGram*) and extend it with the different language fragments. The function *closeGram* closes the constructed grammar and applies the *left-corner transform* in order to remove potential left-recursion; as a consequence straightforward combinator-based top-down parsing techniques can be used in building the parser. Then *generate kws* generates a parser integrated with the semantics for the language starting from the first non-terminal, where the list *kws* is a list of keywords extracted from the grammar description. This takes care of the problem caused by the fact that some identifiers in earlier challenges may become keywords in later challenges. The function *parse* performs the parse of the input program and computes the meaning of that program. In the actual implementation of Oberon0 we generate scanner-less `uu-parsinglib` parsers.

3 Syntax

Using our combinator library `murder`[5] we describe the concrete syntax of each language fragment as a Haskell value. A fragment of the *code constructing the CFG* of the initial language L1 (module *L1.Gram*) is given in Figure 4; the complete definition of the concrete grammar of the four languages can be found in Appendix A. The parameter *sf* contains the "semantics of the language"; its type is defined in the module *L1.Decl* and is derived from the abstract syntax of which we show a fragment in Figure 5. The full abstract syntax of the four languages can be found in Appendix B. We use the Template Haskell function *deriveLang*[6] to derive the type of the

[5] `http://hackage.haskell.org/package/murder`
[6] Provided by the package `AspectAG`.

```
l1 sf = proc _ → do
  rec
    modul ← addNT ≺ ...
    ...
    ss    ← addNT ≺ ⟦ (pSeqStmt sf)
                       stmt
                       (pFoldr (pSeqStmt sf, pEmptyStmt sf)
                               (⟦ ";" stmt ⟧)) ⟧
    stmt  ← addNT ≺ ⟦ (pAssigStmt  sf) ident ":=" exp ⟧
                <|> ⟦ (pIfStmt     sf)
                       "IF" cond
                       (pFoldr (pCondStmtL_Cons sf, pCondStmtL_Nil sf)
                               (⟦ "ELSIF" cond ⟧))
                       mbelse
                       "END" ⟧
                <|> ⟦ (pWhileStmt  sf) "WHILE" exp "DO" ss "END" ⟧
                <|> ⟦ (pEmptyStmt sf) ⟧
    cond  ← addNT ≺ ⟦ (pCondStmt sf) exp "THEN" ss ⟧
    mbelse ← addNT ≺ pMaybe (pMaybeElseStmt_Nothing sf
                            , pMaybeElseStmt_Just sf)
                            (⟦ "ELSE" ss ⟧)
    exp   ← addNT ≺ ...
    ...
  exportNTs ≺ exportList modul $ export cs_Expression     exp
                               . export cs_StmtSeq        ss
                               . export cs_Statement      stmt
                               . export cs_MaybeElseStmt  mbelse
                               . ...
```

Figure 4: Fragment of the concrete syntax specification of L1

record, given the list of data types together composing the abstract syntax tree. For example, for the example fragment we call:

```
$ (deriveLang "L1" [''Module, ''Statement, ''Expression
                   , ''CondStmtL, ''CondStmt, ''MaybeElseStmt])
```

For each production of the abstract syntax tree a field is produced, with name the name of the production prefixed by a *p* and as type the type of the semantic function, which is defined in terms of the semantics associated with the children of the production. For example, the field generated for the production *AssigStmt* is:

```
pAssigStmt :: sf_id_AssigStmt → sf_exp_AssigStmt → sf_AssigStmt
```

For the cases of *List* or *Maybe* type aliases, fields are produced using the name of the non-terminal (i.e. the type) to disambiguate. In our example, for *CondStmtL* we generate the fields *pCondStmtL_Cons* and *pCondStmtL_Nil*, and for *MaybeElseStmt* we generate *pMaybeElseStmt_Just* and *pMaybeElseStmt_Nothing*.

The code of Figure 4 defines the context free grammar of the language fragment, using the record *sf* to add semantics to it. We use the `murder` combinators *pFoldr* and *pMaybe* to model repetition and option, respectively. These combinators are analogous to the respective *foldr* and *maybe* functions.

```
data Statement   = AssigStmt { id_AssigStmt    :: String
                             , exp_AssigStmt   :: Expression }
                 | IfStmt    { if_IfStmt       :: CondStmt
                             , elsif_IfStmt    :: CondStmtL
                             , else_IfStmt     :: MaybeElseStmt }
                 | WhileStmt { exp_WhileStmt   :: Expression
                             , ss_WhileStmt    :: Statement }
                 | SeqStmt   { s1_SeqStmt      :: Statement
                             , s2_SeqStmt      :: Statement }
                 | EmptyStmt
type CondStmtL = [CondStmt]
data CondStmt   = CondStmt { exp_CondStmt :: Expression
                           , ss_CondStmt  :: Statement }
type MaybeElseStmt = Maybe Statement
data Expression = ...
```

Figure 5: AS of the statements of L1

Grammars defined in this way are *extensible*, since further transformations may be applied to the grammar under construction in other modules. Each grammar exports (with *exportNTs*) its starting point (e.g. *modul*) and a table of *exported non-terminals*, each consisting of a label (by convention of the form *cs_...*) and a reference to the current definition of that non-terminal, again a plain Haskell value which can be used and modified in future extensions. Figure 6 contains a fragment of the definition of L2 (from module *L2.Gram*), which extends the L1 grammar with a **FOR**-loop statement. We start by retrieving references to all non-terminals which are to be

```
l2 sf = proc imported → do
  let ss    = getNT cs_StmtSeq    imported
  let stmt  = getNT cs_Statement  imported
  let exp   = getNT cs_Expression imported
  let ident = getNT cs_Ident      imported
  ...
  rec
    addProds ≺ (stmt    , ⟦ (pForStmt sf) "FOR" ident ":=" exp dir exp mbexp
                                           "DO" ss "END" ⟧)
    dir      ← addNT ≺ ⟦ (pTo sf) "TO" ⟧ <|> ⟦ (pDownto sf) "DOWNTO" ⟧
    mbexp    ← addNT ≺ pMaybe (pCst1Exp sf, id) (⟦ "BY" exp ⟧)
    ...
  exportNTs ≺ imported
```

Figure 6: Fragment of the grammar extension L2

extended or used (using *getNT*) from the *imported* non-terminals. We add new productions to existing non-terminals with *addProds*; this does not lead to references to new non-terminals. New non-terminals can still be introduced as well using *addNT*. The Haskell type-system ensures that the *imported* list indeed contains a table with entries *cs_StmtSeq*, *cs_Statement*, *cs_Expression* and *cs_Ident*, and that the types of these non-terminals coincide with their use in the semantic functions of the extensions.

The definition in Figure 6 may look a bit verbose, caused by the interface having been made explicit. Using some Template Haskell this can easily be overcome.

Figure 7 shows the abstract syntax tree fragment corresponding to the **FOR**-loop extension. The prefix *EXT_* indicates that this definition is extending a given non-terminal.

```
data EXT_Statement
   = ForStmt { id_ForStmt :: String, start_ForStmt :: Expression
             , dir_ForStmt :: ForDir, stop_ForStmt :: Expression
             , step_ForStmt :: Expression, ss_ForStmt :: Statement }
   | ...
data ForDir = To | Downto
data EXT_Expression = Cst1Exp
...
```

Figure 7: AST of the **FOR**-loop of L2

4 Aspect Oriented Semantics

The semantics of Oberon0 were implemented using the `AspectAG`[7] embedding of attribute grammars in Haskell. In order to be able to redefine attributes or to add new attributes later, it encodes the lists of inherited and synthesized attributes of a non-terminal as an `HList`-encoded [?] value; each attribute is associated with a unique type which is used as an index in such a "list". The lookup process is performed by the Haskell class mechanism. In this way the *closure test* of the attribute grammar (each attribute has a single definition) is implicitly realised by the Haskell compiler when trying to build the right instances of the classes. Thus, attribute grammar fragments can be individually type-checked, compiled, distributed and composed to construct a compiler.

4.1 Name analysis

Error messages produced by the name analysis are collected in a synthesized attribute called *serr*.The default behaviour of this attribute for most of the productions is to combine (append) the errors produced by the children of the production. This behaviour is captured by the function *use* from the `AspectAG` library, which takes as arguments the label of the attribute to be defined (*serr*), the Haskell list of non-terminals (labels) for which the attribute is defined (*serrNTs*), an operator for combining the attribute values (++), and a unit value to be used when none of the children has such an attribute ([] :: *String*).

```
serrRule = use serr serrNTs (++) ([] :: [String])
```

When a new name is defined we check for multiple declarations and at name uses we check for incorrect uses or uses of undefined identifiers, producing error messages when appropriate. The code below shows the definition of *serr* for the use of an identifier represented by a production *IdExp*, which has a child named *ch_id_IdExp* of type (*DTerm String*)[8].

[7] `http://hackage.haskell.org/package/AspectAG`

[8] *DTerm a* is the type used by murder to represent *attributed terminals* (i.e. identifiers, values); it encodes the value (*value*) and position in the source code (*pos*) of the terminal.

```
serrIdExp = syn serr $ do
  lhs ← at lhs
  nm ← at ch_id_IdExp
  return $ checkName nm (lhs # ienv) ["Var","Cst"] "an expression"
```

With the (plain Haskell) function *checkName* we lookup the name (*nm*) in the symbol table (inherited attribute *ienv* coming from the left-hand side) and, if it is defined, we verify that the name represents either a variable (`"Var"`) or a constant (`"Cst"`) and generate a proper error message if not.

The symbol table is implemented by the pair of attributes *senv* and *ienv*. The synthesized attribute *senv* collects the information from the name declarations and the inherited attribute *ienv* distributes this information through the tree.

In order to perform the name analysis, the type of the symbol table could have been *Map String NameDef*, which is a map from names to values of type *NameDef* representing information about the bound name. However, since we want to use the same symbol table for future extensions, we keep the type "non-closed" by using a list-like structure:

```
data SymbolInfo b a = SI b a
type NMap a = Map String (SymbolInfo NameDef a)
```

For the current task the symbol table includes values of type *NMap a*, parametric in *a*, the "the rest of the information we might want to store for this symbol". In the example below, for declarations of constants, the table consists of a map from the introduced name to a *SymbolInfo* which includes the information needed by the name analysis (constructed using *cstDef*) and some other (yet unknown) information, which is represented by the argument the rule receives:

```
senvCstDecl r = syn senv $ do
  nm ← at ch_id_CstDecl
  return $ Map.singleton (value nm) (SI (cstDef $ pos nm) r)
```

Similarly to how we used *use* for the default cases of synthesized attributes, we capture the behaviour of distributing an inherited attribute to the children of a production with the function *copy*:

```
ienvRule _ = copy ienv ienvNTs
```

The various aspects introduced by the attributes are combined using the function *ext*:

```
aspCstDecl r = senvCstDecl r `ext` ienvCstDecl r `ext` serrCstDecl `ext`
               T1.aspCstDecl
```

In this case, for the production *CstDecl*, we extend *T1.aspCstDecl*, which is imported from *L1.SemT1* and includes the pretty-printing attribute, with the attributes implementing the name analysis task (*serr*, *ienv* and *senv*).

Once the attributes definitions are composed, the semantic functions for the productions may be computed using the function *knit*. For example, the semantic function of the production *CstDecl* in the case of *L1.SemT2* is *knit* (*aspCstDecl* ()). The use of () (unit) here is just to "close the symbol table", since no further information needs to be recorded for Task 2.

4.2 Type checking

Type error messages are collected in the synthesized attribute *sterr*. For type checking we extend the symbol table with the type information (*TInfo*) of the declared names. This is done by *updating* the value of the attribute *senv* with the function *synupdM*, which is similar to *syn* but redefines it making use of its current definition. In the following example we update the symbol table information for the production *VarDecl*, where *sty* is an attribute defined for expressions and types, computing their type information:

$$
\begin{aligned}
&\mathit{senvVarDecl}'\ r = \mathit{synupdM}\ \mathit{senv}\ \$\ \mathbf{do}\\
&\quad \mathit{typ} \leftarrow \mathit{at}\ \mathit{ch\_typ\_VarDecl}\\
&\quad \mathit{return}\ \$\ \mathit{Map.map}\ (\lambda(\mathit{SI}\ \mathit{nd}\ \_) \rightarrow (\mathit{SI}\ \mathit{nd}\ \$\ \mathit{SI}\ (\mathit{typ}\ \#\ \mathit{sty})\ r))
\end{aligned}
$$

The previous definition of the type information is just ignored and only used to indicate the type of the symbol table. Thus, thanks to lazy evaluation, when extending the aspects of Task 2 we only need to pass an undefined value of type *SymbolInfo TInfo a*, where *a* is the type of even further information to be stored in the symbol table (for future extensions):

$$
\begin{aligned}
&\mathit{undTInfo} :: a \rightarrow \mathit{SymbolInfo}\ \mathit{TInfo}\ a\\
&\mathit{undTInfo} = \mathit{const}\ \bot\\
&\mathit{aspVarDecl}\ r = (\mathit{senvVarDecl}'\ r)\ \text{`}\mathit{ext}\text{`}\ \mathit{sterrRule}\ \text{`}\mathit{ext}\text{`}\\
&\qquad\qquad\qquad (\mathit{T2.aspVarDecl}\ \$\ \mathit{undTInfo}\ r)
\end{aligned}
$$

To represent type information we have to deal again with the lack of open data types in Haskell, since we want to keep some specific information for each of the types of the extensible type system we are implementing, and we have decided to resort to the use of Haskell's *Dynamic* type. A *TInfo*, with the information of a certain type, consists of: the representation *trep* of the given type, encapsulated as a *Dynamic* value, a *String* with its pretty-printing (*tshow*), and a function *teq* that, given another type information indicates if the actual type is compatible with the given one.

$$
\begin{aligned}
&\mathbf{data}\ \mathit{TInfo} = \mathit{TInfo}\ \{\mathit{trep} && :: \mathit{Dynamic}\\
&\qquad\qquad\qquad\quad ,\mathit{tshow} && :: \mathit{String}\\
&\qquad\qquad\qquad\quad ,\mathit{teq} && :: (\mathit{TInfo} \rightarrow \mathit{Bool})\}
\end{aligned}
$$

The main task we perform during type checking is to verify whether the actual type of an expression is compatible with the type expected by its context. For example if the condition of an **`IF`** statement has type `BOOLEAN`.

$$
\begin{aligned}
&\mathit{check}\ \mathit{pos}\ \mathit{expected}\ \mathit{got}\\
&\quad = \mathbf{if}\ (\mathit{teq}\ \mathit{expected}\ \mathit{got}) \vee (\mathit{teq}\ \mathit{got}\ \mathit{unkTy}) \vee (\mathit{teq}\ \mathit{expected}\ \mathit{unkTy})\\
&\qquad \mathbf{then}\ [\,]\\
&\qquad \mathbf{else}\ \ [\mathit{show}\ \mathit{pos} + \texttt{": Type error. Expected "} + \mathit{show}\ \mathit{expected} +\\
&\qquad\qquad\quad \texttt{", but got "} + \mathit{show}\ \mathit{got}]
\end{aligned}
$$

If either the expected or the obtained type is unknown (*unkTy*) we do not report a type error, because unknown types are generated by errors that have been already detected by the name analysis process.

A very simple case of type information is the elementary type `BOOLEAN`, where we do not provide any extra information than the type itself. Thus, the type representation is implemented with a singleton type *BoolType*.

$$
\begin{aligned}
&\mathbf{data}\ \mathit{BoolType} = \mathit{BoolType}\\
&\mathit{boolTy} = \mathbf{let}\ d \quad = \mathit{toDyn}\ \mathit{BoolType}\\
&\qquad\qquad\quad\ \ \mathit{bEq} = (\equiv)\ (\mathit{dynTypeRep}\ d)\ .\ \mathit{dynTypeRep}\ .\ \mathit{trep}\ .\ \mathit{baseType}\\
&\qquad\qquad \mathbf{in}\ \ \mathit{TInfo}\ d\ \texttt{"BOOLEAN"}\ \mathit{bEq}
\end{aligned}
$$

To construct the corresponding *TInfo* we convert a *BoolType* value into a *Dynamic* with the function *toDyn*. A type is compatible with `BOOLEAN` if its base type[9] is also `BOOLEAN`, i.e. is compatible if both types are represented with *BoolType* values. With the function *dynTypeRep* we

[9]In case of a user type, the type it denotes.

extract a concrete representation of the type of the value inside a *Dynamic* that provides support for equality.

There exist some other cases were a more involved type representation is needed. For example, in the case of **ARRAY** we include the type information of its elements and the length of the array, if it can be statically computed.

$$\textbf{data}\ \mathit{ArrType} = \mathit{ArrType}\ (\mathit{Maybe\ Int})\ \mathit{TInfo}$$

Then, by using the type-safe cast function *fromDynamic* we can get access to this information provided the dynamic typed value represents an array. Thus, when trying to index a variable, we can for example check if the index is out of range; in case the cast does not succeed we indicate that the variable we are trying to access is not an array:

```
checkSelArray pos ty ind
   = case (fromDynamic . trep . baseType) ty of
     Just (ArrType l _) → checkIndex pos ind l
     _                  → [show pos ++ ": Accessed variable is not an array"]
```

We use the same technique to keep information about the fields of a **RECORD** and the parameters of a **PROCEDURE**.

4.3 Source-to-source transformation

In [?] we extended `AspectAG` with an *agMacro* combinator that enables us to define the attribute computations of a new production in terms of the attribute computations of existing productions. We defined the semantics of the extensions of the language level L2 using this macro mechanism. The **FOR**-loop is implemented as a **WHILE**-loop and the **CASE** statement is defined in terms of an **IF**-**ELSIF**-**ELSE** cascade.

Figure 8 contains the macro definition for the **FOR**-loop, which is parametrized by the attributes (semantics) of:

- *SeqStmt*: sequence of statements
- *AssigStmt*: assign statement
- *IntCmpExp*: integer comparison expression
- *IdExp*: identifier expression
- *IntBOpExp*: integer binary operation expression

We use the combinator *withChildAtt* to obtain the value of the *self* attribute of the child *ch_dir_ForStmt*, with the direction of the iteration. In case the value is *To* the loop counter is incremented (*Plus*) on each step while is less or equal (*LECmp*) the stop value. In other case (*Downto*) we use *Minus* to decrement the counter and *GECmp* (greater or equal) to compare it it with the stop value. In Figure 9 we show the structure of the macro (i.e. the **FOR**-loop in terms of the original AST) for the *To* case. That can be seen as a code translation from:

```
FOR id := start TO stop BY step DO
  ss
END
```

to:

```
id := start;
WHILE id <= stop DO
  ss;
  id := id + step
END
```

```
macroForStmt aspSeqStmt aspAssigStmt aspWhileStmt
             aspIntCmpExp aspIdExp aspIntBOpExp
 = withChildAtt ch_dir_ForStmt self $ λdir →
let (op_stop, op_step) = case dir of
                           To     → (LECmp, Plus)
                           Downto → (GECmp, Minus)
    initStmt   = (aspAssigStmt   ,    ch_id_AssigStmt   ↪ ch_id_ForStmt
                                 <.> ch_exp_AssigStmt  ↪ ch_start_ForStmt)
    whileStmt  = (aspWhileStmt   ,    ch_exp_WhileStmt  ⟹ condWhile
                                 <.> ch_ss_WhileStmt   ⟹ bodyWhile)
    condWhile  = (aspIntCmpExp   ,    ch_op_IntCmpExp   ⇝ op_stop
                                 <.> ch_e1_IntCmpExp   ⟹ idExp
                                 <.> ch_e2_IntCmpExp   ↪ ch_stop_ForStmt)
    idExp      = (aspIdExp       ,    ch_id_IdExp       ↪ ch_id_ForStmt)
    bodyWhile  = (aspSeqStmt     ,    ch_s1_SeqStmt     ↪ ch_ss_ForStmt
                                 <.> ch_s2_SeqStmt     ⟹ stepWhile)
    stepWhile  = (aspAssigStmt   ,    ch_id_AssigStmt   ↪ ch_id_ForStmt
                                 <.> ch_exp_AssigStmt  ⟹ expStep)
    expStep    = (aspIntBOpExp   ,    ch_op_IntBOpExp   ⇝ op_step
                                 <.> ch_e1_IntBOpExp   ⟹ idExp
                                 <.> ch_e2_IntBOpExp   ↪ ch_step_ForStmt)
in  withoutChild ch_dir_ForStmt
         (agMacro (aspSeqStmt  ,    ch_s1_SeqStmt ⟹ initStmt
                               <.> ch_s2_SeqStmt ⟹ whileStmt))
```

Figure 8: Macro definition of the **FOR**-loop

In the cases were specialized behaviour is needed, like for example pretty-printing, it is still possible to redefine the attributes involved on these aspects. As such, our mechanism is much more expressive than conventional macro mechanisms, which only perform a structure transformation. Using the library we get Task 4 almost for free.

Our approach is not very suitable for some other kind of source-to-source transformations like optimizations, because we do not represent the AST with values (if we want to keep the AST extensible) and we (still) do not have higher-order attributes. Although a possible approach is to generate an AST of a fixed core language and perform the optimizations in this language.

4.4 Code generation

We generate the C abstract syntax representation provided by the `language-c`[10] package. This package also includes a pretty-printing function for the abstract syntax.

Since ANSI C does not include nested functions we have to lift all the procedures, types and constants definitions to top-level when generating the C code required by the challenge (note that the lifting as specified is trivial, since the exercise does not require bindings to be lifted properly). In order to avoid name clashes with C keywords or due to the lifting process, we rename every identifier to make it unique. New names are composed by: a character `'_'` (assuring no clashes with C keywords), the path (module and procedure names) to the scope were the name is defined and the actual name. Thus, if we have the following Oberon0 program:

[10] http://hackage.haskell.org/package/language-c

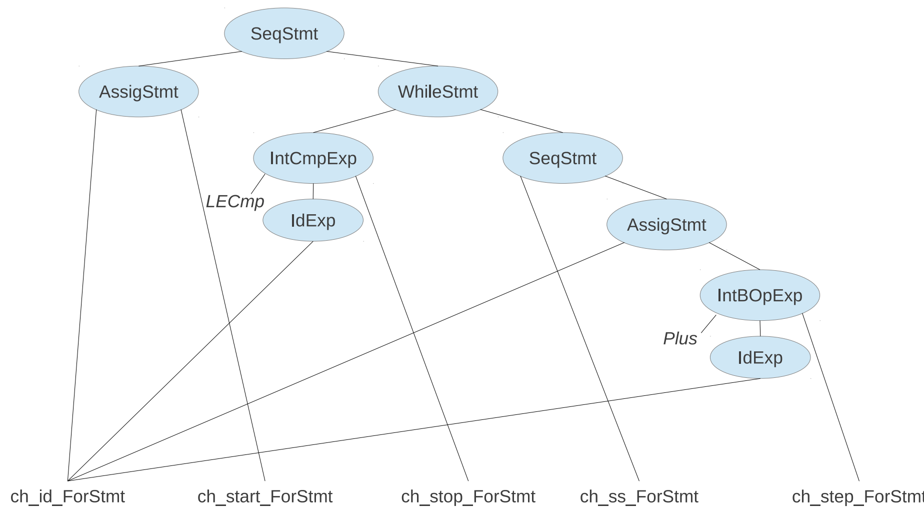

Figure 9: The **FOR**-loop in terms of the original AST

Lang. / Task	*Common*	*T1*	*T2*	*T3*	*T5*	*Total*
Common	-	42	14	-	23	79
L1	128	156	147	220	228	879
L2	187	98	69	65	56	475
L3	94	75	75	134	145	523
L4	48	67	56	197	95	463
Total	457	438	361	616	547	2419

Table 4: Code sizes (in lines of code) of the components of the compiler

```
MODULE A;
  VAR BC : INTEGER;
  PROCEDURE B;
    PROCEDURE C;
    END C
  END B
END A.
```

The names are mapped: the variable name BC to A_BC, the procedure name B to A_B and the procedure name C to A_B_C. Since underscore is not allowed in Oberon0 identifiers, this renaming does not introduce new clashes, like the one we could have had with C if the variable BC was called B_C.

To implement the renaming we extend the symbol table with the name mapping.

5 Artifacts

In Table 4 we show the complexity (in lines of code without comments) of our implementation of the compiler, disaggregated into the different tasks and language levels. The *Common* column includes the *Gram* and *Decl* files, while the *Common* row includes some code used by the *Main* modules.

The code includes 26 lines of Template Haskell, calling functions defined in the libraries to avoid some boilerplate.

We have implemented all the combinations from L1-T1 to L4-T5, including the artifacts proposed by the challenge.

6 Conclusions

The most important aspect of our approach is the possibility to construct a compiler out of a collection of pre-compiled, statically type-checked, possibly mutually dependent language-definition fragments written in Haskell, but with a DSL taste.

When looking at all the aspects we have covered we can conclude that we managed to find solutions for all aspects of the problems; we were rescued by the fact that we could always fall back to plain Haskell, in case our libraries were not providing a standard solution for the problem at hand. We have seen such solutions for dealing with flexible symbol tables, generating new identifiers and types.

We mention again that our implementation is quite verbose, since each module contains quite some code "describing its interface" in the collection of co-operating modules. This is the price we have to pay for getting the extreme degree of flexibility we are providing. By collapse the modules the amount of linking information shrinks considerably. Other option to reduce verbosity is to use `uuagc` to generate `AspectAG` code [?].

Another cause of the verbosity is that we have not used the system itself or Template Haskell to capture some common patterns. We have chosen to reveal the underlying mechanisms, the role of the type system, the full flexibility provided, and have left open the possibility for further extensions.

The lack of open data types in Haskell makes it hard to implement AST transformations in extensible languages using our technique. Semantic macros solve some of these problems. A possible approach is to use our technique to implement the front-end of a compiler, translating to a core fixed language, and then use other more traditional approaches (like `uuagc`) to implement the back-end. Another option is to use *data types à la carte* [?] to simulate open data types (and functions) in Haskell.

References

[1] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Proc. of the 2004 Workshop on Haskell*, pages 96–107. ACM Press, 2004.

[2] Wouter Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, July 2008.

[3] Marcos Viera and S. Doaitse Swierstra. Attribute grammar macros. In *XVI Simpósio Brasileiro de Linguagens de Programação*, LNCS, pages 150–165, 2012.

[4] Marcos Viera, S. Doaitse Swierstra, and Atze Dijkstra. Grammar Fragments Fly First-Class. In *Proc. of the 12th Workshop on Language Descriptions Tools and Applications*, pages 47–60, 2012.

[5] Marcos Viera, S. Doaitse Swierstra, and Arie Middelkoop. UUAG Meets AspectAG. In *Proc. of the 12th Workshop on Language Descriptions Tools and Applications*, 2012.

[6] Marcos Viera, S. Doaitse Swierstra, and Wouter Swierstra. Attribute Grammars Fly First-Class: How to do aspect oriented programming in Haskell. In *Proc. of the 14th Int. Conf. on Functional Programming*, pages 245–256, New York, USA, 2009. ACM.

[7] Niklaus Wirth. *Compiler construction*. International computer science series. Addison-Wesley, 1996.

A Concrete Grammar

A.1 L1

```
$ (csLabels ["cs_Module", "cs_Declarations", "cs_Expression", "cs_Factor"
            , "cs_StmtSeq", "cs_Statement", "cs_MaybeElseStmt"
```

```
        ,"cs_Ident","cs_IdentL","cs_Type"])

l1 sf = proc _ → do
  rec
    modul     ← addNT ≺ ⟦ (pModule sf)
                           "MODULE" ident ";"
                           decls
                           (pMaybe (pEmptyStmt sf, id) (⟦ "BEGIN" ss ⟧))
                           "END" ident "." ⟧
    decls     ← addNT ≺ ⟦ (pDeclarations sf)
                           (pMaybe (pDeclL_Nil sf, id) (⟦ "CONST" cstDeclL ⟧))
                           (pMaybe (pDeclL_Nil sf, id) (⟦ "TYPE" typDeclL ⟧))
                           (pMaybe (pDeclL_Nil sf, id) (⟦ "VAR"  varDeclL ⟧)) ⟧
    cstDeclL  ← addNT ≺ pFoldr  (pDeclL_Cons sf, pDeclL_Nil sf)
                                 (⟦ (pCstDecl sf) ident "=" exp ";" ⟧)
    typDeclL  ← addNT ≺ pFoldr  (pDeclL_Cons sf, pDeclL_Nil sf)
                                 (⟦ (pTypDecl sf) ident "=" typ ";" ⟧)
    varDeclL  ← addNT ≺ pFoldr  (pDeclL_Cons sf, pDeclL_Nil sf)
                                 (⟦ (pVarDecl sf) idL ":" typ ";" ⟧)
    idL       ← addNT ≺ ⟦ (pIdentL_Cons sf) ident
                           (pFoldr (pIdentL_Cons sf, pIdentL_Nil sf)
                                   (⟦ "," ident ⟧)) ⟧
    typ       ← addNT ≺ ⟦ (pType sf) ident ⟧
    exp       ← addNT ≺ ⟦ sexp ⟧
                   <|> ⟦ (eExp   sf) exp "="  sexp ⟧
                   <|> ⟦ (neExp  sf) exp "#"  sexp ⟧
                   <|> ⟦ (lExp   sf) exp "<"  sexp ⟧
                   <|> ⟦ (leExp  sf) exp "<=" sexp ⟧
                   <|> ⟦ (gExp   sf) exp ">"  sexp ⟧
                   <|> ⟦ (geExp  sf) exp ">=" sexp ⟧
    sexp      ← addNT ≺ ⟦ signed ⟧
                   <|> ⟦ (plusExp  sf) sexp "+"  signed ⟧
                   <|> ⟦ (minusExp sf) sexp "-"  signed ⟧
                   <|> ⟦ (orExp    sf) sexp "OR" signed ⟧
    signed    ← addNT ≺ ⟦ term ⟧
                   <|> ⟦ (posExp sf) "+" term ⟧ <|> ⟦ (negExp sf) "-" term ⟧
    term      ← addNT ≺ ⟦ factor ⟧
                   <|> ⟦ (timesExp sf) term "*"   factor ⟧
                   <|> ⟦ (divExp   sf) term "DIV" factor ⟧
                   <|> ⟦ (modExp   sf) term "MOD" factor ⟧
                   <|> ⟦ (andExp   sf) term "&"   factor ⟧
    factor    ← addNT ≺ ⟦ (trueExp  sf) (kw "TRUE")  ⟧
                   <|> ⟦ (falseExp sf) (kw "FALSE") ⟧
                   <|> ⟦ (pParExp  sf) "(" exp ")"  ⟧
                   <|> ⟦ (notExp   sf) "~" factor   ⟧
                   <|> ⟦ (pIdExp sf) ident ⟧ <|> ⟦ (pIntExp sf) int ⟧
    ss        ← addNT ≺ ⟦ (pSeqStmt sf) stmt
                                         (pFoldr (pSeqStmt sf, pEmptyStmt sf)
                                                 (⟦ ";" stmt ⟧)) ⟧
    stmt      ← addNT ≺ ⟦ (pAssigStmt  sf) ident ":=" exp ⟧
```

```
                          <|> ⟦ (pIfStmt       sf)
                                "IF" cond
                                (pFoldr (pCondStmtL_Cons sf, pCondStmtL_Nil sf)
                                        (⟦ "ELSIF" cond ⟧))
                                mbelse
                                "END" ⟧
                          <|> ⟦ (pWhileStmt  sf) "WHILE" exp "DO" ss "END" ⟧
                          <|> ⟦ (pEmptyStmt sf) ⟧
    cond      ← addNT ≺ ⟦ (pCondStmt sf) exp "THEN" ss ⟧
    mbelse    ← addNT ≺ pMaybe ( pMaybeElseStmt_Nothing sf
                               , pMaybeElseStmt_Just sf)
                               (⟦ "ELSE" ss ⟧)
    ident     ← addNT ≺ ⟦ var ⟧ <|> ⟦ con ⟧
  exportNTs ≺ exportList modul $ export cs_Declarations  decls
                               . export cs_Expression    exp
                               . export cs_Factor        factor
                               . export cs_StmtSeq       ss
                               . export cs_Statement     stmt
                               . export cs_Ident         ident
                               . export cs_IdentL        idL
                               . export cs_MaybeElseStmt mbelse
                               . export cs_Type          typ
```

A.2 L2

```
l2 sf = proc imported → do
  let ss     = getNT cs_StmtSeq       imported
  let stmt   = getNT cs_Statement     imported
  let exp    = getNT cs_Expression    imported
  let ident  = getNT cs_Ident         imported
  let mbelse = getNT cs_MaybeElseStmt imported
  rec
    addProds ≺ (stmt    ,  ⟦ (pForStmt   sf) "FOR" ident ":=" exp dir exp mbexp
                                             "DO" ss "END" ⟧
                       <|> ⟦ (pCaseStmt sf) "CASE" exp "OF"
                                             c cs mbelse "END" ⟧)
    dir       ← addNT ≺ ⟦ (pTo sf) "TO" ⟧ <|> ⟦ (pDownto sf) "DOWNTO" ⟧
    mbexp     ← addNT ≺ pMaybe (pCst1Exp sf, id) (⟦ "BY" exp ⟧)
    cs        ← addNT ≺ pFoldr (pCaseL_Cons sf, pCaseL_Nil sf) (⟦ "|" c ⟧)
    c         ← addNT ≺ ⟦ (pCase sf) labels ":" ss ⟧
    labels    ← addNT ≺ ⟦ (pLabelL_Cons sf) label
                          (pFoldr (pLabelL_Cons sf, pLabelL_Nil sf)
                                  (⟦ "," label ⟧)) ⟧
    label     ← addNT ≺ ⟦ (pExpreLbl sf) exp ⟧
                    <|> ⟦ (pRangeLbl sf) exp ".." exp ⟧
  exportNTs ≺ imported
```

A.3 L3

```
l3 sf = proc imported → do
  let decls = getNT cs_Declarations imported
```

```
  let stmt  = getNT cs_Statement    imported
  let ss    = getNT cs_StmtSeq      imported
  let exp   = getNT cs_Expression   imported
  let ident = getNT cs_Ident        imported
  let idl   = getNT cs_IdentL       imported
  let typ   = getNT cs_Type         imported
  rec
    addProds  ≺ (stmt,       ⟦ (pProcCStmt sf) ident params ⟧)
    params    ← addNT  ≺  ⟦ "(" paraml ")" ⟧ <|> ⟦ (pExpressionL_Nil sf) ⟧
    paraml    ← addNT  ≺  ⟦ (pExpressionL_Cons sf) exp
                            (pFoldr (pExpressionL_Cons sf, pExpressionL_Nil sf)
                                    (⟦ "," exp ⟧)) ⟧
                      <|> ⟦ (pExpressionL_Nil sf) ⟧
    updProds  ≺ (decls, λdeclarations → ⟦ (pExtDeclarations sf) declarations
                                                              procDeclL ⟧)
    procDeclL ← addNT  ≺  pFoldr (pDeclL_Cons′ sf, pDeclL_Nil′ sf)
                                 (⟦ procDecl ⟧)
    procDecl  ← addNT  ≺  ⟦ (pProcDecl sf) "PROCEDURE" ident fparams ";"
                                           decls
                                           (pMaybe (pEmptyStmt′ sf, id)
                                                   (⟦ "BEGIN" ss ⟧))
                                           "END" ident ";" ⟧
    fparams   ← addNT  ≺  ⟦ "(" fparaml ")" ⟧ <|> ⟦ (pParamL_Nil sf) ⟧
    fparaml   ← addNT  ≺  ⟦ (pParamL_Cons sf) fparam
                            (pFoldr (pParamL_Cons sf, pParamL_Nil sf)
                                    (⟦ ";" fparam ⟧)) ⟧
                      <|> ⟦ (pParamL_Nil sf) ⟧
    fparam    ← addNT  ≺  ⟦ (fpVar  sf) "VAR" idl ":" typ ⟧
                      <|> ⟦ (fpVal  sf)       idl ":" typ ⟧
  exportNTs ≺ imported
```

A.4 L4

```
l4 sf = proc imported → do
  let stmt   = getNT cs_Statement  imported
  let exp    = getNT cs_Expression imported
  let factor = getNT cs_Factor     imported
  let ident  = getNT cs_Ident      imported
  let idl    = getNT cs_IdentL     imported
  let typ    = getNT cs_Type       imported
  rec
    addProds ≺ (typ       ,  ⟦ (pArrayType  sf) "ARRAY" exp "OF" typ ⟧
                         <|> ⟦ (pRecordType sf) "RECORD" fieldl "END" ⟧)
    fieldl    ← addNT  ≺  ⟦ (pFieldL_Cons sf) field
                            (pFoldr  (pFieldL_Cons sf, pFieldL_Nil sf)
                                     (⟦ ";" field ⟧)) ⟧
    field     ← addNT  ≺  ⟦ (pField sf) idl ":" typ ⟧ <|> ⟦ (pEmptyField sf) ⟧
    addProds ≺ (factor,      ⟦ (pSelExp sf) ident selector ⟧)
    selector  ← addNT  ≺  ⟦ (pSelectL_Cons sf) sel
                            (pFoldr  (pSelectL_Cons sf, pSelectL_Nil sf)
```

```
                                              (⟦ sel ⟧)) ⟧
    sel        ← addNT  ≺  ⟦ (pSelField sf ) "." ident ⟧
                          <|> ⟦ (pSelArray sf )  "[" exp "]" ⟧
    addProds ≺ (stmt,        ⟦ (pAssigSelStmt sf ) ident selector ":=" exp ⟧)
  exportNTs ≺ imported
```

B Abstract Syntax

B.1 L1

```
data Module = Module { idbgn_Module :: String
                     , decls_Module  :: Declarations
                     , stmts_Module :: Statement
                     , idend_Module :: String }

data Declarations = Declarations { cstdecl_Declarations :: DeclL
                                 , typdecl_Declarations :: DeclL
                                 , vardecl_Declarations :: DeclL }

type DeclL = [Decl]

data Decl = CstDecl  { id_CstDecl  :: String, exp_CstDecl  :: Expression }
          | TypDecl { id_TypDecl  :: String, typ_TypDecl :: Type }
          | VarDecl { idl_VarDecl :: IdentL, typ_VarDecl :: Type }

data Type = Type { id_Type :: String }

data Statement =   AssigStmt  { id_AssigStmt    :: String
                              , exp_AssigStmt  :: Expression }
               |   IfStmt     { if_IfStmt         :: CondStmt
                              , elsif_IfStmt      :: CondStmtL
                              , else_IfStmt       :: MaybeElseStmt }
               |   WhileStmt { exp_WhileStmt :: Expression
                              , ss_WhileStmt   :: Statement }
               |   SeqStmt    { s1_SeqStmt      :: Statement
                              , s2_SeqStmt      :: Statement }
               |   EmptyStmt

type CondStmtL = [CondStmt]

data CondStmt = CondStmt { exp_CondStmt :: Expression
                         , ss_CondStmt   :: Statement }

type MaybeElseStmt = Maybe Statement

type IdentL = [String]

type GHC_IntCmp = IntCmp
data IntCmp = ECmp | NECmp | LCmp | LECmp | GCmp | GECmp
type GHC_IntBOp = IntBOp
data IntBOp = Plus | Minus | Times | Div | Mod
type GHC_IntUOp = IntUOp
data IntUOp = Ng | Ps

type GHC_BoolBOp = BoolBOp
data BoolBOp = Or | And
type GHC_BoolUOp = BoolUOp
data BoolUOp = Not

data Expression = IntCmpExp   { op_IntCmpExp   :: GHC_IntCmp
                              , e1_IntCmpExp   :: Expression
```

```
                  , e2_IntCmpExp   :: Expression}
    | IntBOpExp   { op_IntBOpExp   :: GHC_IntBOp
                  , e1_IntBOpExp   :: Expression
                  , e2_IntBOpExp   :: Expression}
    | IntUOpExp   { op_IntUOpExp   :: GHC_IntUOp
                  , e_IntUOpExp    :: Expression}
    | BoolBOpExp  { op_BoolBOpExp  :: GHC_BoolBOp
                  , e1_BoolBOpExp  :: Expression
                  , e2_BoolBOpExp  :: Expression}
    | BoolUOpExp  { op_BoolUOpExp  :: GHC_BoolUOp
                  , e_BoolUOpExp   :: Expression}
    | IdExp       { id_IdExp       :: String}
    | IntExp      { int_IntExp     :: Int}
    | BoolExp     { bool_BoolExp   :: Bool}
    | ParExp      { e_ParExp       :: Expression}

$ (deriveAG "Module)
$ (deriveLang "L1" ["Module, "Declarations, "DeclL, "Decl, "Type
                   , "Statement, "CondStmtL, "CondStmt, "MaybeElseStmt
                   , "Expression, "IdentL])

eExp  sf = pIntCmpExp sf (sem_Lit ECmp)
neExp sf = pIntCmpExp sf (sem_Lit NECmp)
lExp  sf = pIntCmpExp sf (sem_Lit LCmp)
leExp sf = pIntCmpExp sf (sem_Lit LECmp)
gExp  sf = pIntCmpExp sf (sem_Lit GCmp)
geExp sf = pIntCmpExp sf (sem_Lit GECmp)
plusExp  sf = pIntBOpExp sf (sem_Lit Plus)
minusExp sf = pIntBOpExp sf (sem_Lit Minus)
timesExp sf = pIntBOpExp sf (sem_Lit Times)
divExp   sf = pIntBOpExp sf (sem_Lit Div)
modExp   sf = pIntBOpExp sf (sem_Lit Mod)
posExp sf = pIntUOpExp sf (sem_Lit Ps)
negExp sf = pIntUOpExp sf (sem_Lit Ng)
orExp  sf = pBoolBOpExp sf (sem_Lit Or)
andExp sf = pBoolBOpExp sf (sem_Lit And)
notExp sf = pBoolUOpExp sf (sem_Lit Not)
trueExp  sf t = pBoolExp sf (λr → DTerm (pos (t r)) True)
falseExp sf f = pBoolExp sf (λr → DTerm (pos (f r)) False)
```

B.2 L2

```
data EXT_Statement
  = ForStmt  { id_ForStmt :: String, start_ForStmt :: Expression
             , dir_ForStmt :: ForDir, stop_ForStmt :: Expression
             , step_ForStmt :: Expression, ss_ForStmt :: Statement}
  | CaseStmt { exp_CaseStmt :: Expression, case_CaseStmt :: Case
             , cases_CaseStmt :: CaseL, else_CaseStmt :: MaybeElseStmt}
data ForDir = To | Downto
type CaseL = [Case]
data Case = Case { label_Case :: LabelL, ss_Case :: Statement}
```

```
type LabelL = [Label]
data Label = ExpreLbl  { exp_ExpreLbl :: Expression }
           | RangeLbl { e1_RangeLbl  :: Expression
                      , e2_RangeLbl  :: Expression }
data EXT_Expression = Cst1Exp
```

```
$ (extendAG ''EXT_Statement [''Statement, ''MaybeElseStmt, ''Expression])
$ (extendAG ''EXT_Expression [])
$ (deriveLang "L2" [''EXT_Statement, ''ForDir, ''CaseL, ''Case
                   , ''LabelL, ''Label, ''EXT_Expression])
```

B.3 L3

```
type GHC_KindParam = KindParam
data KindParam = VarP | ValP
data Param = Param { kind_Param :: GHC_KindParam
                   , idl_Param  :: IdentL
                   , typ_Param  :: Type }
type ParamL = [Param]
data EXT_Decl = ProcDecl { id_ProcDecl     :: String
                         , params_ProcDecl :: ParamL
                         , decls_ProcDecl  :: Declarations
                         , stmts_ProcDecl  :: Statement
                         , idend_ProcDecl  :: String }
data EXT_Declarations
          = ExtDeclarations  { decls_ExtDeclarations   :: Declarations
                             , prcdecl_ExtDeclarations :: DeclL }
type ExpressionL = [Expression]
data EXT2_Statement = ProcCStmt { id_ProcCStmt     :: String
                                , params_ProcCStmt :: ExpressionL }
$ (extendAG ''EXT_Decl [''Declarations, ''Statement, ''IdentL, ''Type])
$ (extendAG ''EXT_Declarations [''Declarations, ''DeclL])
$ (extendAG ''EXT2_Statement [''Expression])
$ (deriveLang "L3" [''EXT_Declarations, ''EXT_Decl, ''Param, ''ParamL
                   , ''EXT2_Statement, ''ExpressionL])
```

B.4 L4

```
data EXT_Type = ArrayType  { exp_ArrayType :: Expression
                           , typ_ArrayType :: Type }
              | RecordType { fields_RecordType :: FieldL }
type FieldL = [Field]
data Field = Field { idl_Field :: IdentL, typ_Field :: Type }
           | EmptyField
data EXT2_Expression = SelExp { id_SelExp :: String, sel_SelExp :: SelectL }
type SelectL = [Select]
data Select = SelField  { id_SelField   :: String }
            | SelArray { exp_SelArray :: Expression }
data EXT3_Statement = AssigSelStmt { id_AssigSelStmt  :: String
```

$$
\begin{aligned}
&\qquad\qquad\qquad\qquad\qquad\qquad , \mathit{sel\_AssigSelStmt} \; :: \mathit{SelectL} \\
&\qquad\qquad\qquad\qquad\qquad\qquad , \mathit{exp\_AssigSelStmt} :: \mathit{Expression}\} \\
&\$\,(\mathit{extendAG}\ \text{``}\mathit{EXT\_Type}\ [\text{``}\mathit{Expression}, \text{``}\mathit{IdentL}, \text{``}\mathit{Type}]) \\
&\$\,(\mathit{extendAG}\ \text{``}\mathit{EXT2\_Expression}\ [\text{``}\mathit{Expression}]) \\
&\$\,(\mathit{extendAG}\ \text{``}\mathit{EXT3\_Statement}\ [\text{``}\mathit{SelectL}, \text{``}\mathit{Expression}]) \\
&\$\,(\mathit{deriveLang}\ \texttt{"L4"}\ [\text{``}\mathit{EXT\_Type}, \text{``}\mathit{FieldL}, \text{``}\mathit{Field}, \text{``}\mathit{EXT2\_Expression} \\
&\qquad\qquad\qquad\quad , \text{``}\mathit{SelectL}, \text{``}\mathit{Select}, \text{``}\mathit{EXT3\_Statement}])
\end{aligned}
$$