POET Tutorial: Building A Small Compiler Project

Qing Yi
University of Colorado at Colorado Springs
(qyi@cs.uccs.edu)

September 12, 2019

A compiler is essentially a translator that reads in some code in one language and produces a result in another language. The following demonstrates basic techniques to accomplish parsing, AST construction, type checking, unparsing, and basic program analysis when using POET to implement a small compiler project.

1 Parsing

The following illustrates a simple parser (in POET/examples/compiler_1.pt) in the POET language.

```
<parameter in message="input file name"/>
<* parse the in File using syntax defined in file "compiler_input_1.code" *>
<input from=(in) syntax="compiler_input_1.code" />
```

Here the first line declares a command-line parameter named *in*, and the second line is a POET comment that explains the third line.

The following Linux command invokes the above translator to parse the "exp.input" file contained in the same directory.

```
> pcg -pin=exp.input compiler_1.pt
```

Here the command-line option $-pin = exp.input$ defines exp.input as the value for the parameter *in* declared at line 1 of compiler1.pt. Alternatively, the following Linux command accomplishes the same purpose.

```
> pcg compiler_1.pt < exp.input
```

Here since the value for the command-line parameter *in* is undefined, the input is read from the standard input (the keyboard), for which the content is supplied from file exp.input using Linux shell redirection.

2 Writing Syntax Specifications For Parsing

The *input* command at line 3 of the compiler_1.pt file is used to parse an arbitrary file using the concrete syntax defined in the *compiler_input_1.code* file, which aims to support the following BNF specifications.

```
Block ::= For | E ; | Decl | Block Block
For ::= for ( id = E ; id < E ; E)  { Block }
E ::=  E + F | E [ E ]  | F = E | F ++
F ::= id | intval | floatval
Decl ::= Type id ;
Type ::= int | float [ Type ]
```

After eliminating left-recursion (POET parses the input code in a top-down recursive descent fashion, so left-recursion in the grammar needs to be eliminated) and applying left factoring, we obtain the following BNF.

```
Block ::= For | E ; | Decl | Block Block
For ::= for ( id = E ; id < E ; E)  { Block }
E ::=  F E'
E' ::= + F E' | [ E ] E' |  = E |  ++ | epsilon
F ::= id | intval | floatval
Decl ::= Type id ;
Type ::= int | float | [ Type ]
```

The following is a POET implementation of the above BNF in POET/examples/compiler_input_1.code.

```
<define PARSE CODE.Block/>
<define TOKEN (("+" "+") ("-" "-")) />
<define KEYWORDS ("for" "int" "float")/>

<code Block parse=LIST(CODE.For | (CODE.Exp ";") | CODE.Decl, "\n") />
<code For parse=("for" "(" ID "=" CODE.Exp ";" ID "<" CODE.Exp ";" CODE.Exp ")" "{" CODE.Block "}")/>
<code Exp parse=(CODE.Factor CODE.Exp1) />
<code Exp1 parse=(("+" CODE.Factor CODE.Exp1)
                |("[" CODE.Exp "]" CODE.Exp1)
                |("=" CODE.Exp)
                |"++"
                |"") />
<code Factor parse=(ID | INT | FLOAT) />
<code Decl parse=(CODE.Type ID ";")/>
<code Type parse=("int"|"float"|("[" Type "]"))/>
```

Here a one-to-one correspondence can be clearly seen from the POET implementation to the original BNF specification. The first line of the POET implementation defines the start non-terminal to be Block. The second line extends the POET internal tokenizer to recognize two additional tokens, "++" and "–". The third line extends the POET internal tokenizer to recognize three keywords, "for", "int", and "float", instead of treating them as identifiers. Finally, each production x ::= β is translated to <code x parse=(β)/>, where each non-terminal name x is referenced using $CODE.x$ within the *parse* phrase. Each of the *code* definitions is called a **code template** in POET. The three keywords, ID, INT, and FLOAT, are built-in token types used by POET to represent identifiers, integers, and floating point numbers respectively. The POET built-in *LIST* operator is used in the production for *Block* to easily specify a sequence of items separated by a single token (a line break).

3 Type Checking

To apply type checking to an input program, a type must be uniquely determined for each expression within the input program, by recording a unique type declared for each variable through a sequence of nested symbol tables. A translation scheme for type checking can be specified like the following.

Block ::= For | E ';' | Decl | Block Block
For ::= for '(' id '=' E^1 ';' id '<' E^2 ';' E^3 ')' '{' { push_scope(); } Block { pop_scope(); } '}'
    { t1:=Lookup_Variable(id); if (t1$\neq$IntType or t1 $\neq$ E^1.type or t1 $\neq$ E^2.type) ERROR; }
E ::= F { E'.inh := F.type; } E' { E.type := E'.type; }
E' ::= '+' F { if (E'.inh $\neq$ F.type or (E'.inh$\neq$IntType|FloatType)) ERROR;
    E'^1.inh := E'.inh; } E'^1 { E'.type:=E'^1.type; }
  | '[' E ']' { if (E.type $\neq$ IntType or E'.inh $\neq$ PtrType(t1)) ERROR;
    E'^1.inh:=t1; } E'^1 { E'.type := E'^1.type; }
  | '=' E { if (E'.inh $\neq$ E.type) ERROR; E'.type := E.inh; }
  | '++' { if (E'.inh $\neq$ IntType) ERROR; E'.type:=IntType; }
  | ϵ { E'.type := E'.inh; }
F ::= id { F.type = Lookup_Variable(id); if (F.type=empty) ERROR; }
  | intval { F.type = IntType; }
  | floatval { F.type := FloatType; }
Decl ::=Type id ; { if (Lookup_Variable(id.name) ERROR; insert_entry(id.name, Type.type); }
Type ::=int { Type.type := IntType; } | float { Type.type = FloatType; }
  | '[' $Type^1$ ']' { Type.type = PtrType($Type^1$.type); }

Combined with global symbol table management, the above translation scheme can be implemented by augmenting the original syntax specifications in POET/examples/compiler_input_1.code with the following (POET/examples/compiler_input_2.code)

```
<define PARSE CODE.Block/>
<define TOKEN (("+" "+") ("-" "-")) />
<define KEYWORDS ("for" "int" "float")/>

<******************************************************************>
<* enumeration of different types, each a subtype of Type *>
<******************************************************************>
<code IntType match=CODE.Type/>
<code FloatType match=CODE.Type/>
<code TypeError match=CODE.Type/>

<* PtrType is a struct with a single member variable: elemtype*>
<code PtrType pars=(elemtype) match=CODE.Type/>

<******************************************************************>
<* Parsing and Translation schemes for type checking  *>
<******************************************************************>
<code Block parse=LIST(CODE.For | (CODE.Exp ";") | CODE.Decl, "\n") />
<code For parse=("for" "(" (id=ID) "=" (e1=CODE.Exp) ";" ID "<" (e2=CODE.Exp) ";" CODE.Exp ")"
                 "{" eval(push_scope()) CODE.Block eval(pop_scope()) "}"
                 eval(t1=LookupVariable(id);
                 if (t1=="") print("Line " LINE_NO ": Undefined variable:" id);
                 if (t1 != IntType || t1 != e1 || t1 != e2)
                   print("Line " LINE_NO ": type mismatch:" id "vs." e1 "vs." e2);) )/>
<code Exp parse=(CODE.Factor (e1=CODE.Exp1) eval(return(e1)))/>
<code Exp1 parse=(("+" (e1=CODE.Factor)
              eval(if (INHERIT==IntType && e1==IntType) { IntType }
                  else if (INHERIT==FloatType && e1==FloatType) { FloatType }
                  else { print("Line " LINE_NO ": mismatching type: " INHERIT "vs." e1); CODE.TypeError})
         (e2=CODE.Exp1) eval(return(e2)))
```

```
       |  ("[" (sub=CODE.Exp) "]"
          eval(if (sub!=IntType) print("Line " LINE_NO ": array subscript must be an integer: " sub);
               if (!(INHERIT : PtrType#(t1))) { t1=TypeError;
                  print("Line " LINE_NO ": expecting an array type but get:" INHERIT); }
          t1) (e1=CODE.Exp1) eval(return(e1)))
       |  ("=" (e1=CODE.Exp)
          eval(if (INHERIT!=e1) print("Line " LINE_NO ": mismatching type:" INHERIT "vs." e1); return(INHERIT)))
       |  ("++" eval(if (INHERIT!=IntType)
             print("Line " LINE_NO ": cannot apply ++ to " INHERIT); return IntType; ))
       |  ("" eval(return(INHERIT)))) />
<code Factor parse=((id=ID) eval(t=LookupVariable(id);
       if (t=="") print("Line " LINE_NO ": undefined variable:" id); return(t)))
      | (INT eval(return IntType))
      | (FLOAT eval(return FloatType)) />
<code Decl parse=((t=CODE.Type) (id=ID) ";"
                  eval(if (LookupVariable(id)) ERROR("Variable already defined:" id); insert_entry(id, t);)) />
<code Type parse=( ("int" eval(return IntType))
               |   ("float" eval(return FloatType))
               |   ("[" (t=Type) "]" eval(return(PtrType#(t)))))/>
```

The translation scheme is implemented through syntax-directed translation by inserting *eval* invocations inside the productions inside the *parse* attributes. The above example demonstrates the use of if-else conditionals as well as assignments and function calls to implement the necessary type checking support. Finally, if a synthesized attribute needs to be returned as result of the parsing function, the value of the attribute is used as parameter to invoke the *return* operation inside the final *eval* invocation of the production. The *print* operation is invoked to print out an error message without stopping execution of the parser. The LINE_NO macro is used to extract the line number of the input file currently being processed. Note that the $E1$ non-terminal has an inherited attribute named *inh*, and its definition is specified using the $inh = INHERIT$ attribute within the $E1$ specification.

Note that when defining a compound type *PtrType*, we used the following.

```
<code PtrType pars=(elemtype) />
```

The above definition essentially defines a C struct type which has a single member variable named *elemtype*. To build an object of the struct, use syntax *PtrType#v*, where v is the value of *elemtype*. To check whether a variable x is a PtrType, use notation x : *PtrType#t*, where t is an undefined variable. If x is indeed a PtrType, this pattern matching expression returns true and then initializes t so that it contains the value of elemtype of x; otherwise, the pattern matching returns false. For more details, please reference the POET language manual.

POET provides a compound data type, MAP, to support the need of associative tables that map arbitrary keys to their values. The following POET statement constructs an empty symbol table for our simple toy language and stores the table into a global variable *symTable*.

```
<define symTable MAP(ID,"int"|"float"|"") />
```

If nested scopes are allowed, a list of symbol tables can be used to represent the context information of each block. Specifically, the global variable *symTable* is used to map each variable name to its declared type. The function *LookupVariable* is used to look up variable information from the symbol table. The following code from *POET/examples/compiler_2.pt* illustrates the use and definition of the symbol table.

```
<parameter infile message="input file name"/>

<** a global variable named symTable is used to save all variable info**>
<define symTable MAP(ID,"int"|"float"|"") />

<***************************************************************>
<* parse the in File using syntax defined in file "compiler_input_2.code";
   save the parsing result (the abstract syntax tree) to ast variable *>
<input from=(infile) syntax="compiler_input_2.code" to=ast/>

<* look up a variable name from a list of symble tables*>
<xform LookupVariable pars=(varName)  symTableList=GLOBAL.symTable >
  for ( p = symTableList; p != NULL; p = TAIL(p)) {
     curTable = HEAD(p);
     res = curTable[varName];
     if (res != "") RETURN res;
  }
</xform>

<xform insert_entry pars=(id,type) symTableList=GLOBAL.symTable >
  (HEAD(symTableList))[id]=type;
  ""
</xform>

<xform pop_scope >
  top = HEAD(GLOBAL.symTable);
  GLOBAL.symTable = TAIL(GLOBAL.symTable);
  top
</xform>

<xform push_scope>
  GLOBAL.symTable = MAP{} :: GLOBAL.symTable;
</xform>

<* print out the symbol table content *>
<eval print(symTable) />
```

4 Constructing The AST

A real compiler needs to save the input program after parsing, typically in the form of the Abstract Syntax Tree (AST). In POET, AST construction is supported through the concept of *code templates*, which are used in the parsing phase to recognize the structure of the input code, in the program evaluation phase to represent the internal structure of programs, and in the unparsing phase to output results to external files. So far we have used the POET code template to specify how to parse an input language, where the **name** of each *code template* corresponds to the left-hand non-terminal of a BNF production, and its **parse attribute** corresponds to the right-hand side of the production. For example, the following code template definition corresponds to the BNF production *E ::= F E1.*

```
<code E parse=(CODE.F CODE.E1) />
```

In addition to being used to parse an input language, POET code templates can also be used to specify the internal representation, specifically the Abstract Syntax Representation, of an input program. For example, the following code template definition additionally specifies what needs to be saved in the BNF production *E ::= F E1*.

```
<code E pars=(opd1 : CODE.F, rest : CODE.E1 ) parse=(opd1 rest) />
```

In particular, if the value of any symbol in the right-hand side of a BNF production needs to be saved, a corresponding variable can be created as a parameter of the POET *code template*, and the variable is used in place of the original symbol in the right hand of the production (i.e., inside the parse attribute) to save the targeted value. Note that the type of each member variable x is declared using the $x : t$ notation, where t defines the code template type of x so that x can be parsed correctly when used in the right hand side of E's production.. Here two variables, $opd1$ and $rest$, are created and used to replace the non-terminals F and $E1$ in the original production respectively. The above definition essentially defines a AST node type (similar to a C struct type) which is named E and has two member variables named opd1 and rest respectively.

To build an object of the struct, use syntax $E\#(v1, v2)$, where v1 and v2 are the values of $opd1$ and $rest$ respectively. To check whether a variable contains a value of the AST node type E, use notation $x : E\#(t1, t2)$, where $t1$ and $t2$ are undefined variables used to save the values of $opd1$ and $rest$ if x is indeed an E AST node type, in which case the overall pattern matching expression returns true; otherwise the pattern matching returns false. For more details, please reference the POET language manual. The following code template definition shows how to save the parsing result as an object of the corresponding AST node type.

```
<code E pars=(opd1:CODE.F, rest:CODE.E1 ) parse=(opd1 rest eval(return(E#(opd1,rest)))) />
```

Without the eval/return invocation, the parsing result (synthesized attribute of the production) would have simply been a singly-linked list that contains two components, opd1 and rest. The notation $E\#(opd1, rest)$ builds an AST node type named E, which is internally equivalent to a struct type in C with two data members named opd1 and rest respectively. Explicitly building AST node types allows the internal structure of the AST IR to be tested a lot more explicitly and thus enables much more effective processing of the AST.

As a practical example, the following extends the type checking translation scheme in Section 3 to additionally construct an AST IR as the result of parsing. Note that in order to support both type checking and AST construction, we have defined two synthesized attributes for each expression E, E.type and E.ast, which are used to save the type and the AST IR of the expression respectively.

Block ::=For { Block.ast:=For.ast; } | E ';' { Block.ast:=make_expStmt(E.ast,E.type); }
    | Decl { Block.ast = Decl.ast; } | Block^1 Block^2 { Block.ast=make_block(Block^1.ast,Block^2.ast); }
For ::= for '(' id '=' E^1 ';' id '<' E^2 ';' E^3 ')' '{' { push_scope(); } Block { pop_scope(); } '}'
    { t1:=Lookup_Variable(id); if (t1$\neq$IntType or t1 $\neq$ E^1.type or t1 $\neq$ E^2.type) ERROR;
      For.ast=make_forLoop(id.name, E^1.ast, E^2.ast,E^3); }
E ::= F { E'.inh := (F.ast,F.type); } E' { E.type := E'.type; E.ast=E'.ast; }
E' ::= '+' F { if (E'.inh.type $\neq$ F.type or (E'.inh.type$\neq$IntType|FloatType)) ERROR;
    E'^{1}.inh := (make_bop("+",E'.inh.ast,F.ast),F.type); } E'^{1} { E'.type:=E'^{1}.type; E'.ast:=E'^{1}.ast; }
  | '[' E ']' { if (E.type $\neq$ IntType or E'.inh.type $\neq$ PtrType(t1)) ERROR;
    E'^{1}.inh:=(make_arrAcc(E'.inh.ast,E.ast),t1);} E'^{1} { E'.type := E'^{1}.type; E'.ast:=E'^{1}.ast; }
  | '=' E { if (E'.inh.type $\neq$ E.type) ERROR;
        E'.type := E'.inh.type; E'.ast=make_bop("=", E'.inh.ast,E.ast); }
  | '++' { if (E'.inh.type $\neq$ IntType) ERROR; E'.type:=IntType; E'.ast=make_uop("++",E'.inh.ast); }

    | ϵ { E'.type := E'.inh.type; E'.ast=E'.inh.ast; }
F ::= id { F.type = Lookup_Variable(id.name); F.ast=id.name; if (F.type=empty) ERROR; }
    | intval { F.type = IntType; F.ast=intval.value; }
    | floatval { F.type := FloatType; F.ast = floatval.value; }
Decl ::=Type id ; { if (Lookup_Variable(id.name) ERROR;
                        insert_entry(id.name, Type.type); Decl.ast=make_decl(Type.type,id.name); }
Type ::=int { Type.type := IntType; } | float { Type.type = FloatType; }
        | '[' Type^1 ']' { Type.type = PtrType(Type^1.type); }

The above translation scheme is implemented in POET/examples/*compiler_input_3.code*, the content of which is shown in the following.

```
<define PARSE CODE.Block/>
<define TOKEN (("+" "+") ("-" "-")) />
<define KEYWORDS ("for" "int" "float")/>

<*****************************************************************>
<* enumeration of different types, each a subtype of Type *>
<*****************************************************************>
<code IntType match=CODE.Type/>
<code FloatType match=CODE.Type/>
<code TypeError match=CODE.Type/>

<* PtrType is a struct with a single member variable: elemtype*>
<code PtrType pars=(elemtype) match=CODE.Type/>

<*****************************************************************>
<* ATTR implements the synthesized/inherited attribute of each expression*>
<* It is a struct with two member variables: exp and type *>
<*****************************************************************>
<code ATTR pars=(exp,type) />

<* representation of the whole program *>
<code Block pars=(program:LIST(CODE.For | CODE.ExpStmt | CODE.Decl, "\n"))
  parse=(program eval(return(Block#program))) />

<* ExpStmt is a type of AST node and is a a struct with two member variables: exp and its type*>
<code ExpStmt pars=(exp : CODE.Exp, type)
  parse=(exp ";" eval(return(ExpStmt#(exp[ATTR.exp],exp[ATTR.type]))))/>

<* For is a type of AST node and is a struct with 5 member variables: var,lb,ub,incr, body*>
<code For pars=(var:ID, lb:CODE.Exp, ub:CODE.Exp, incr:CODE.Exp, body:CODE.Block)
  parse=("for" "(" var "=" lb ";" var "<" ub ";" incr ")" "{" eval(push_scope()) body eval(pop_scope()) "}"
          eval(t1=LookupVariable(var); if (t1=="") print("Undefined variable:" var);
               if (t1 != lb[ATTR.type] || t1 != CODE.IntType) print("Type mismatch:" var " vs. " lb);
               if (t1 != ub[ATTR.type]) print("Type mismatch:" var " vs. " ub);
               return(For#(var,lb[ATTR.exp],ub[ATTR.exp],incr[ATTR.exp],body)) )) />

<* Decl is a type of AST node and is a struct with two member variables: t and id *>
<code Decl pars=(t : CODE.Type, id : ID)
   parse=(t id ";" eval(if (LookupVariable(id)) ERROR("Variable already defined:" id);
                        insert_entry(id, t); return(Decl#(t,id)))) />

<* two types of AST nodes for expressions, each a subtype of Exp *>
<code Bop pars=(op, opd1, opd2) match=CODE.Exp/>
<code Uop pars=(op, opd) match=CODE.Exp/>
```

```
<******************************************************************>
<* parse the types and expressions and return the corresponding AST*>
<******************************************************************>

<code Exp parse=(CODE.Factor (e1=CODE.Exp1) eval(return(e1)))/>

<code Exp1 parse=(("+" (e1=CODE.Factor)
                  eval(if (INHERIT[ATTR.type]==IntType && e1[ATTR.type]==IntType) t=CODE.IntType;
                  else if (INHERT[ATTR.type]==FloatType && e1[ATTR.type]==FloatType) t = CODE.FloatType;
                  else { print("Mismatching type: " INHERIT "vs." e1); t = CODE.TypeError; }
                  ATTR#(Bop#("+",INHERIT[ATTR.exp],e1[ATTR.exp]),t))
        (e2=CODE.Exp1) eval(return(e2)))
      |  ("[" (sub=CODE.Exp) "]"
         eval(if (sub[ATTR.type]!=IntType) print("Array subscript must be an integer: " sub);
              if (!(INHERIT[ATTR.type] : PtrType#(t1))) { t1=TypeError; print("Not an array type:" INHERIT); }
              ATTR#(Bop#("[]",INHERIT[ATTR.exp],sub[ATTR.exp]),t1))
         (e2=CODE.Exp1) eval(return(e2)))
      |  ("=" (e1=CODE.Exp)
            eval(if (INHERIT[ATTR.type]!=e1[ATTR.type]) print("Mismatching type: " INHERIT "vs." e1);
                 return (CODE.ATTR#(Bop#("=",INHERIT[ATTR.exp],e1[ATTR.exp]), INHERIT[ATTR.type]))))
      |  ("++" eval(if (INHERIT[ATTR.type]!=IntType) print("Cannot apply ++ to " INHERIT);
                  return (CODE.ATTR#(Uop#("++",INHERIT[ATTR.exp]),IntType))))
      |  ("" eval(return(INHERIT)))) />

<code Factor parse=(((id=ID) eval(t = LookupVariable(id); if (t=="") print("Undefined variable:" id);
                                 return(CODE.ATTR#(id,t))))
      | ((v=INT) eval(return (CODE.ATTR#(v,IntType))))
      | ((v=FLOAT) eval(return (CODE.ATTR#(v,FloatType))))) />

<code Type parse=( ("int" eval(return IntType)) | ("float" eval(return FloatType))
                  |("[" (t=Type) "]" eval(return(PtrType#(t)))))/>
```

In the above, two additional *code templates*, *Bop* and *Uop*, are created as new types of the AST nodes to save the binary and unary operations in an expression. A code template, ATTR, is created to save both the AST and the type of an expression inside a single struct. These three code templates are used only as internal representations of the input program. Specifically, they are not used in parsing, so they do not have to have their parse attributes defined. The *match* attribute of Bop and Uop are used to define their subtype relations to the code template Exp. Several other code templates, e.g., *For*, *Block*, and *Decl*, are extended with parameters (member variables) to remember the key components of their productions.

When using the *compiler_3.pt* file to parse and print out the internal representation of the following input code (POET/examples/compiler.input),

```
int i;
int b;
[float] c;
for (i = 0; i < b+10; i++) { b = b + 1; c[i]=0.1; }
```

the following AST will be printed out.

```
Block#Decl#(
    IntType,
    "i")
  Decl#(
    IntType,
    "b")
  Decl#(
    PtrType#FloatType,
    "c")
  For#(
    "i",
    0,
    Bop#(
      "+",
      "b",
      10),
    Uop#(
      "++",
      "i"),
    Block#ExpStmt#(
        Bop#(
          "=",
          "b",
          Bop#(
            "+",
            "b",
            1)),
        IntType)
      ExpStmt#(
        Bop#(
          "=",
          Bop#(
            "[]",
            "c",
            "i"),
          "0.1"),
        FloatType)
       NULL)
   NULL
```

Note that each AST node is built (printed out) using the *op#(p1,...,pm)* notation, where *op* is the code template name, and $p1, ..., pm$ are values for the code template parameters.

5 Unparsing AST

After parsing and saving the parsed input program as an AST, the AST representation eventually needs to be unparsed back into strings of readable syntax. The following shows the unparsing specification for the for loop in C. In the following example,

```
<code Loop pars=(i:ID,start:EXP, stop:EXP, step:EXP) >
for (@i@=@start@; @i@<@stop@; @i@+=@step@)
</code>
```

Here the list of strings between the pair of <code...> and </code> defines the **body** of the *Loop* code template. Each template body is typically a list of strings in a POET input/output language such as C, C++. POET expressions, e.g., the template parameters i, *start*, *stop*, and *stop* in the above example, need to be wrapped inside pairs of @s within the body to be properly evaluated. Dynamic code generation is supported by embedding POET expressions inside the pairs of @s as needed. The formatting of the unparsing is preserved as much as possible when unparsing the IR.

The following POET driver script in POET/examples/compiler_4.pt shows how to use POET to parse and unparse an input program.

```
<parameter infile message="input file name"/>

<define symTable ""/>

<* parse the in File using syntax defined in file "compiler_input_4.code";
   save the parsing result (the abstract syntax tree) to ast variable *>
<input from=(infile) syntax="compiler_input_4.code" to=ast />

<output from=(ast) syntax="compiler_input_4.code" />
```

The following POET implementation from POET/examples/compiler_input_4.code extends that in Section 4 to additionally unparse the IR with proper syntax.

```
<define PARSE CODE.Goal/>
<define TOKEN (("+" "+") ("-" "-")) />
<define KEYWORDS ("for" "int" "float")/>

<******* a global variable named symTable is used to save all variable info*****>
<define symTable MAP(ID,"int"|"float"|"") />

<* look up a variable name from a list of symble tables*>
<xform LookupVariable pars=(varName)  symTableList=GLOBAL.symTable >
  for ( p = symTableList; p != NULL; p = TAIL(p)) {
     curTable = HEAD(p);
     res = curTable[varName];
     if (res != "") RETURN res;
  }
</xform>

<xform insert_entry pars=(id,type) symTableList=GLOBAL.symTable >
  (HEAD(symTableList))[id]=type;
  ""
</xform>


<******************************************************************>

<* enumeration of different types, each a subtype of Type *>
<code IntType match=CODE.Type> int </code>
<code FloatType match=CODE.Type> float </code>
<code TypeError match=CODE.Type> TYPE_ERROR </code>

<* PtrType is a struct with a single member variable: elemtype*>
<code PtrType pars=(elemtype) match=CODE.Type> [@elemtype@] </code>

<******************************************************************>
<* ATTR implements the synthesized/inherited attribute of each expression*>
<* It is a struct with two member variables: exp and type *>
<******************************************************************>
<code ATTR pars=(exp,type) />

<* representation of the whole program *>
<code Goal pars=(program:LIST(CODE.For | CODE.ExpStmt | CODE.Decl, "\n")) >
@program@
</code>
```

```
<* ExpStmt is a type of AST node and is a a struct with two member variables: exp and its type*>
<code ExpStmt pars=(exp : CODE.Exp, type)
  parse=(exp ";" eval(return(ExpStmt#(exp[ATTR.exp],exp[ATTR.type]))))>
@exp@;
</code>

<* For is a type of AST node and is a struct with 5 member variables: var,lb,ub,incr, body*>
<code For pars=(var:ID, lb:CODE.Exp, ub:CODE.Exp, incr:CODE.Exp, body:CODE.Goal)
  parse=("for" "(" var "=" lb ";" var "<" ub ";" incr ")" "{" body "}"
          eval(t1=LookupVariable(var);
               if (t1=="") print("Undefined variable:" var);
               if (t1 != lb[ATTR.type] || t1 != CODE.IntType)
                  print("Type mismatch:" var " vs. " lb);
               if (t1 != ub[ATTR.type]) print("Type mismatch:" var " vs. " ub);
               return(For#(var,lb[ATTR.exp],ub[ATTR.exp],incr[ATTR.exp],body)) )) >
for (@var@=@lb@; @var@ < @ub@; @incr@)
{
  @body@
}

</code>

<* Decl is a type of AST node and is a struct with two member variables: t and id *>
<code Decl pars=(t : CODE.Type, id : ID)
   parse=(t id ";" eval(if (LookupVariable(id)) ERROR("Variable already defined:" id);
                     insert_entry(id, t); return(Decl#(t,id))))>
@id@ : @t@;
</code>

<* two types of AST nodes for expressions, each a subtype of Exp *>
<code Bop pars=(op, opd1, opd2) match=CODE.Exp> (@opd1 op opd2@) </code>
<code Uop pars=(op, opd) match=CODE.Exp> @op opd@ </code>

<***************************************************************>
<* parse the types and expressions and return the corresponding AST*>
<***************************************************************>

<code Type parse=( ("int" eval(return IntType))
                 |("float" eval(return FloatType))
                 |("[" (t=Type) "]" eval(return(PtrType#(t)))))/>

<code Exp parse=(CODE.Factor (exp1=CODE.Exp1) eval(return(exp1)))/>

<code Exp1
  parse=(("+" (factor=CODE.Factor)
         eval(
           if (INHERIT[ATTR.type]==IntType && factor[ATTR.type]==IntType)
                {t=CODE.IntType;}
           else if (INHERT[ATTR.type]==FloatType && factor[ATTR.type]==FloatType)
                { t = CODE.FloatType; }
           else { print("mismatching type: " INHERIT "vs." factor); t = CODE.TypeError; }
           ATTR#(Bop#("+",INHERIT[ATTR.exp],factor[ATTR.exp]),t))
        (exp1=CODE.Exp1) eval(return(exp1)))
      |("[" (sub=CODE.Exp) "]"
            eval(if (sub[ATTR.type]!=IntType) print("Array subscript must bean integer: " sub);
                 if (INHERIT[ATTR.type] : PtrType#(t1)) { t=t1; } else { t=TypeError; }
                 ATTR#(Bop#("[]",INHERIT[ATTR.exp],sub[ATTR.exp]),t))
         (exp1=CODE.Exp1) eval(return(exp1)))
      |("=" (exp1=CODE.Exp)
            eval(if (INHERIT[ATTR.type]!=exp1[ATTR.type])
                    print("mismatching type: " INHERIT "vs." exp1);
                 return (CODE.ATTR#(Bop#("=",INHERIT[ATTR.exp],exp1[ATTR.exp]),
                                    INHERIT[ATTR.type]))))
      |("++" eval(if (INHERIT[ATTR.type]!=IntType) print("Cannot apply ++ to " INHERIT);
                  return (CODE.ATTR#(Uop#("++",INHERIT[ATTR.exp]),IntType))))
      | ("" eval(return(INHERIT)))) />

<code Factor
  parse=(((id=ID) eval(t = LookupVariable(id);
                       if (t=="") print("Undefined variable:" id);
                       return(CODE.ATTR#(id,t))))
          | ((v=INT) eval(return (CODE.ATTR#(v,IntType))))
          | ((v=FLOAT) eval(return (CODE.ATTR#(v,FloatType))))) />
```

6 Three-address code generation

This means translating the input program to another lower-level language. Here the symbol table needs to remember the memory allocation decisions for all variables, a set of new code templates need to be defined to implement the output language, and a translation scheme similar to the following needs to be implemented.

```
Block ::= {For.begin=Block.begin; } For  { Block.next = For.next; Block.instr=For.instr; }
       | {E.begin=Block.begin; } E {Block.next=empty; Block.instr=E.instr; }
       | {Decl.begin=Block.begin; } Decl  { Block.next=empty; Block.instr=empty; }
       | {Block1.begin=Block.begin; } Block1 {Block2.begin=Block1.next;} Block2
         { Block.next=Block2.next; Block.instr=append(Block1.instr, Block2.instr); }
Decl ::= Type id ; { manage_variable_memory(id, Type.size); }
Type ::= int { Type.size=2; } | float { Type.size=4; }
For ::= for ( id = {E1.begin=For.begin;} E1 ; id < {E2.begin=new_label();} E2 ;
        {E3.begin=Block.next; } E3 )  {Block.begin=new_label();} Block
        { r = new_reg(); offset=Lookup(id);
          s1 = gen_3addr(empty,"store",E1.reg,"rarp",offset);
          s2 = gen_3addr(empty,"loadAI","rarp",offset,r);
          s3 = gen_3addr(empty,"if<",r, E2.reg, body_label);
          For.next = new_label();
          s4 = gen_3addr(empty,"goto",empty,empty, For.next);
          s5 = gen_3addr(empty, "goto", empty, empty, E2.begin);
          For.instr = append(E1.instr,s1,E2.instr,s2,s3,s4,Block.instr,E3.instr,s5); }
E ::= {E1.begin=E.begin; } E1 + { E2.begin=empty; } E2
   { E.reg = new_reg(); E.instr=gen_3addr(empty, "add",E1.reg,E2.reg,E.reg); }
  | id = { E2.begin=E.begin; } E2
   { E.reg = E2.reg;
     E.instr=append(E2.instr,gen_3addr(empty,"storeAI",E2.reg,"rarp",Lookup(id))); }
  |  id { E.reg = new_reg();
        E.instr=gen_3addr(E.begin,"loadAI","rarp",Lookup(id),E.reg); }
  | ++ id { E.reg = new_reg(); offset=Lookup(id);
          E.instr=gen_3addr(E.begin,"loadAI","rarp",offset, E.reg);
          E.instr=gen_3addr(empty,"addi",E.reg, 1, E.reg);
          E.instr=gen_3addr(empty,"storeAI",E.reg,"rarp",offset); }
  | -- id { E.reg = new_reg(); offset=Lookup(id);
          E.instr=gen_3addr(E.begin,"loadAI","rarp",offset, E.reg);
          E.instr=gen_3addr(empty,"addi",E.reg, 1, E.reg);
          E.instr=gen_3addr(empty,"storeAI",E.reg,"rarp",offset); }
  | intval { E.reg = new_reg();
             gen_3addr(E.begin,"loadi", intval, empty, E.reg); }
  | floatval { E.reg = new_reg();
             gen_3addr(E.begin,"loadi",floatval, empty, E.reg); }
```

Here the following attributes are defined for the non-terminals.

```
 Block.begin, For.begin, E.begin:
     inherited attributes used to save the beginning label of the corresponding 3-address instructions;
 Block.next, For.next:
     synthesized attributes used to save the beginning label for the next following statement;
 Block.instr, For.instr, E.instr:
     the actual 3-address instructions generated for each AST node;
 E.reg:
     the register used to save the result of each expression
```

All the attributes must be evaluated (defined) at the appropriate places. Note that the above translation scheme is not an L-attributed definition because the inherited attribute E3.begin in the For production uses the *next* synthesized attribute of the loop body. So the scheme cannot be implemented correctly during parsing. However, it can be easily implemented correctly by translating the loop body before E3 when traversing the AST. The following illustrates a POET implementation of the above translation scheme (POET/examples/compiler_5.pt) together with some simple memory management and output language definitions.

```
<parameter infile message="input file name"/>
<parameter outfile message="output file name"/>

<*========== reading the input code ============*>
<* parse the in File using syntax defined in file "compiler_input_4.code";
   save the parsing result (the abstract syntax tree) to ast variable *>
<input from=(infile) syntax="compiler_input_4.code" to=ast/>

<*============== three-address code management =====*>
<define ir_3addr NULL/> <* resulting the 3-address IR generated *>
<define localOffset 16/> <* beginning offset of local variables *>
<define reg_index 0/>  <* index of new registers created *>
<define label_index 0/> <* index of new labels created *>

<* a function that returns a new register *>
<xform new_register > GLOBAL.reg_index=GLOBAL.reg_index+1; "r_"^reg_index </xform>

<* a function that returns a new label *>
<xform new_label > GLOBAL.label_index=GLOBAL.label_index+1; "L_"^label_index </xform>

<* 3-address IR date structure *>
<code ThreeAddress pars=(label, op, opd1, opd2, opd3)>
```

```
@label@: @op@ @opd1@ @opd2@ @opd3@
</code>

<code InstrList pars=(content)>
@((TAIL(content) : NULL)? "" : InstrList#(TAIL(content)))
HEAD(content)@

</code>


<* a function for appending a new 3-address instruction*>
<xform Append_instr pars=(label, op, opd1, opd2, opd3)>
GLOBAL.ir_3addr = ThreeAddress#(label, op, opd1, opd2, opd3) :: GLOBAL.ir_3addr;
</xform>

<*=============== translation schemes for 3-addr code generation =========*>
<* implement translation scheme for all productions of non-terminal Goal *>
<xform Gen3Address_Goal pars=(input, begin)>
  CODE.Goal#(program)=input;
  for (p=program; p != NULL; p = cdr(p)) {
    switch (cur=car(p)) {
      case CODE.For : begin=XFORM.Gen3Address_For(cur,begin);
      case CODE.Decl : XFORM.Gen3Address_Decl(cur,begin); begin="";
      case CODE.ExpStmt#(CLEAR e,CLEAR t) :
                  XFORM.Gen3Address_Exp(e,t,begin); begin="";
    }
  }
  begin
</xform>

<* implement translation scheme for all productions of non-terminal Decl *>
<xform Gen3Address_Decl pars=(input,begin)>
  Decl#(type,name) = input;
  switch (type) {
  case CODE.IntType:
     XFORM.insert_entry(name,GLOBAL.localOffset);
     localOffset = localOffset + 2;
  case CODE.FloatType | CODE.PtrType:
     XFORM.insert_entry(name,GLOBAL.localOffset);
     localOffset = localOffset + 4;
  }
  begin
</xform>

<* implement translation scheme for all productions of non-terminal For *>
<xform Gen3Address_For pars=(input,begin)>
  For#(var,lb,ub,incr,body) = input;
  r_var = new_register();
  offset = XFORM.LookupVariable(var);
  if (offset == "") ERROR("undefined variable:" var);
  r_exp1 = XFORM.Gen3Address_Exp(lb,CODE.IntType,begin);
  Append_instr("", "store", r_exp1, "rarp", offset);
  test_label = new_label();
  body_label = new_label();
  exit_label = new_label();
  r_exp2 = XFORM.Gen3Address_Exp(ub,CODE.IntType,test_label);
  Append_instr("", "loadAI", "rarp", offset, r_var);
  Append_instr("", "if<", r_var, r_exp2, body_label);
  Append_instr("", "goto", "", "", exit_label);
  body_end_label = XFORM.Gen3Address_Goal(body,body_label);
  XFORM.Gen3Address_Exp(incr, CODE.IntType, body_end_label);
  Append_instr("", "goto", "", "", test_label);
  exit_label
</xform>

<code LHSType pars=(basetype) />
<* implement translation scheme for all productions of non-terminal Exp *>
<xform Gen3Address_Exp pars=(exp, type, begin) >
  switch(exp)
  {
    case CODE.Bop#("+", exp1, exp2):
          t1 = Gen3Address_Exp(exp1,type,begin);
          t2 = Gen3Address_Exp(exp2,type,"");
          t3 = new_register();
          Append_instr("", "add", t1, t2, t3);
          t3
    case CODE.Bop#("[]", exp1, exp2):
          t2 = Gen3Address_Exp(exp2,CODE.IntType,"");
          if (!(type : LHSType#(elemtype))) elemtype=type;
          t3 = new_register();
          Append_instr("", "multi", t2, ((elemtype==IntType)?2:4), t3);
          (t1,s1) = Gen3Address_Exp(exp1,LHSType#elemtype,begin);
          Append_instr("", "add", t1, t3, t3);
          if (!(type : CODE.LHSType)) { Append_instr(begin, "loadAI", t3, s1, t3); t3}
          else  { (t3, s1) }
    case CODE.Bop#("=", var, exp2):
          t2 = Gen3Address_Exp(exp2,type,begin);
          (base,offset) = Gen3Address_Exp(var,LHSType#type,"");
          Append_instr("","storeAI",t2,base,offset);
          t2
     case ID:
         offset=LookupVariable(exp);
         if (!(type : LHSType)) {
            t1 = new_register();
            Append_instr(begin, "loadAI", "rarp", offset, t1);
            t1
         }
         else { ("rarp",offset) }
     case CODE.Uop#("++",var):
         t1 = new_register(); offset=LookupVariable(var);
         Append_instr(begin, "loadAI", "rarp", offset, t1);
         Append_instr("", "addi", t1, 1, t1);
         Append_instr("", "storeAI",t1, "rarp", offset);
         t1
     case INT | FLOAT:
         t1 = new_register();
         Append_instr(begin, "loadi", exp, "", t1);
         t1
  }
</xform>

<*==================== driver code ===================*>
<* start evaluation *>
<eval last = Gen3Address_Goal(ast, "");
```

```
      if (last != "") Append_instr(last, "", "", "", ""); <* generate the last label *>
/>

<* write the 3-address ir to the given output file*>
<output from=(InstrList#ir_3addr) to=outfile />
```

In above implementation, instead of saving all the 3-address instructions incrementally as a synthesized attribute, a global variable named *ir_3addr* is used to save the whole sequence of instruction generated, and the code generation process for each AST node is carefully coordinated to ensure all the 3-address instructions are appended at the end of the global sequence in the correct order. Otherwise, the AST traversal functions are quite similar in structure to the type checking code in POET/examples/compiler_4.pt. Each traversal function includes the inherited attribute of its non-terminal as an input parameter and returns the synthesized attribute of its non-terminal as result. Finally the resulting three-address code is unparsed with proper syntax using the following POET output command.

```
 <output to=(out)  from=(InstrList#ir_3addr) />
```

Here the *InstrList* and the *ThreeAddress* code templates define both the internal representation and the concrete syntax of a linear sequence of three-address instructions.

7 Building A Control Flow Graph

A control flow graph can also built while traversing the AST representation of an input program, by promptly identifying the ending of old basic blocks, the beginning of new basic blocks, and connecting them properly. Then the graph can be output using code template syntax definitions for the nodes and edges. The construction algorithm can be specified using the following translation scheme.

```
Block ::= {For.begin=Block.begin; } For  { Block.next = For.next; Block.cfg=For.cfg; }
      | {E.begin=Block.begin; } E {Block.next = append(Block.next, E.ast);  }
      | {Decl.begin=Block.begin; } Decl  { Block.next=append(Block.next,Decl.ast); }
      | {Block1.begin=Block.begin; } Block1 {Block2.begin=Block1.next;} Block2
        { Block.next=Block2.next; Block.cfg=combine(Block1.cfg, Block2.cfg); }
For ::= for ( id = E1 ; id <  E2 ;
        {E3.begin=Block.next; } E3 )  {Block.begin=new_label();} Block
       { For.next = new_label();
         prev = append(For.begin, assign(id,E1.ast));
         test = Bop#("<", id, E2.ast);
         body=append(Block.next,E3.ast);
         For.cfg=combine(Block.cfg, flow#(prev->test),
                          flow#(test->body),flow#(test->For.next), flow#(body,test));}
```

Here only the Block and For non-terminals need to be processed. The following attributes are defined for the non-terminals.

```
 Block.begin, For.begin
     inherited attributes used to save the current basic block being processed.
 Block.next, For.next:
     synthesized attributes used to save the basic block to be expanded by the following statement;
 Block.cfg, For.cfg:
     the actual control flow graph generated for the non-terminals;
```

Note that both the definitions and the uses of the attributes are quite similar to those for three-address code generation in compiler_5.pt. The following illustrates a POET implementation of the above translation scheme at POET/examples/compiler_6.pt.

```
<parameter infile message="input file name"/>
<parameter outfile message="output file name"/>

<*========== reading the input code ============*>
<* parse the in File using syntax defined in file "compiler_input_4.code";
   save the parsing result (the abstract syntax tree) to ast variable *>
<input from=(infile) syntax="compiler_input_4.code" to=ast/>

<*============== control flow graph IR and management =====*>
<define cfg_nodes NULL/> <* resulting CFG nodes generated *>
<define cfg_edges NULL/> <* resulting CFG edges generated *>
<define label_index 0/>  <* index of new basic blocks created *>

<* a function that returns a new label *>
<xform new_label > GLOBAL.label_index=GLOBAL.label_index+1; GLOBAL.label_index </xform>

<* basic block IR date structure *>
<code BasicBlock pars=(label, stmts)>
B@label@[@CODE.print_list#(stmts,"")@]
</code>

<* CFG edge IR date structure *>
<code Flow pars=(from, to)>
B@from@->B@to@
</code>

<code CFG pars=(nodes, edges)>
digraph CFG
{
  @CODE.print_list#(nodes,"\n")@
  @CODE.print_list#(edges,"\n")@
}

</code>
<* print out the content backward *>
<code print_list pars=(content,sep)>
@((TAIL(content) : NULL)? HEAD(content) :
  (print_list#(TAIL(content),sep) sep HEAD(content)))@
</code>

<* a function for generate a new basic block *>
<xform new_basicblock pars=(stmts)>
label = GLOBAL.label_index;
GLOBAL.cfg_nodes = BasicBlock#(label, stmts) :: GLOBAL.cfg_nodes;
GLOBAL.label_index = GLOBAL.label_index + 1;
label
</xform>

<* a function for generate a new cfg edge *>
```

```
<xform new_flow pars=(from, to)>
GLOBAL.cfg_edges = Flow#(from, to) :: GLOBAL.cfg_edges;
</xform>
<****translation schemes for control flow graph contruction **************>
<* implement translation scheme for all productions of non-terminal Goal *>
<xform BuildCFG__Goal pars=(input, begin)>
  CODE.Goal#(block) = input;
  for (p = block; p != NULL; p = TAIL(p)) {
    cur = HEAD(p);
    switch (cur) {
    case CODE.For : begin = XFORM.BuildCFG__For(cur,begin);
    case CODE.Decl : begin = cur :: begin;
    case CODE.ExpStmt: begin=cur :: begin;
    }
  }
  begin
</xform>

<* implement translation scheme for all productions of non-terminal For *>
<xform BuildCFG__For pars=(input,begin)>
  For#(var,lb,ub,incr,body) = input;
  b_init = new_basicblock(CODE.Bop#("=",var,lb) :: begin); <* wrap up the previous basic block *>
  b_test = new_basicblock(Bop#("<",var,ub)); <* new basic block for test *>
  body_label= GLOBAL.label_index; <* label for the first basic block of body *>
  body_next = XFORM.BuildCFG__Goal(body,""); <* lable of the last basic block from loop body *>
  body_last = new_basicblock(incr::body_next); <* last basic block from loop body*>
  exit_label = new_label(); <* exit label for the loop*>
  <* new generate cfg edges *>
  new_flow(b_init, b_test);
  new_flow(b_test, body_label);
  new_flow(b_test, exit_label);
  new_flow(body_last, b_test);
  "" <* the exit block is currently empty*>
</xform>

<*===================== driver code ===================*>
<* start evaluation *>
<eval last = BuildCFG__Goal(ast, "");
      new_basicblock(""); <* generate the last basic block *>
/>

<* write the 3-address ir to the given output file*>
<output from=(CFG#(cfg_nodes, cfg_edges)) syntax="compiler_input_4.code" to=outfile />
```

8 Traversing The Control-flow Graph

Building a control-flow graph simply means identifying all the basic blocks and then adding edges to represent the flow of control among the basic blocks. However, the control-flow graph often needs to be traversed in efficient ways. For example, to support data flow analysis, we need to quickly find all the successors (or predecessors) of each basic block, by using additional data structures, e.g., associative maps. The following code builds an associative table that maps each basic block to a list of its successors.

```
<xform MapSuccessors pars=(cfg)>
  res = MAP(_,_);
  foreach edge = Flow#(CLEAR from, CLEAR to) \in cfg do
     res[from] = BuildList(to, res[from]);
  enddo
  res
</xform>
```

9 Data-flow Analysis

When performing data-flow analysis, each basic block needs to be associated with a set of information (e.g., a set of expressions or variables). These sets of information can be implemented using either the built-in compound type *lists* in POET or using associative maps in POET. In general, you need to first traverse the entire input, find the set of all the entities in your analysis domain. The set of information associated with each basic block is then a subset of this domain.

10 Modifying the AST

The *REPLACE* operator is the most useful operation in POET for supporting code transformations. It can be invoked using two different syntax, illustrated in the following.

```
    REPLACE("x","y", SPLIT("","x*x-2"));
    REPLACE( (("a",1) ("b",2) ("c",3)), SPLIT("","a+b-c"));
```

The first REPLACE invocation replaces all occurrences of string “x” with string “y” in the third argument (the result of $SPLIT(\text{“”}, \text{“}x * x - 2\text{”})$). In contrast, the second REPLACE invocation makes a sequence of replacements one after another: it first looks for the occurrence of string “a” to replace it with 1; then it tries to locate string “b” to replace it with 2, before looking for “c”, and so on. In the second invocation, each replacement is applied only once, and a warning message is issued if the entire list of replacement specifications are not exhausted when reaching the end of the last argument of REPLACE.

Note that both REPLACE operation perform the modifications by returning a new data structure instead of modifying the original input code. The following is a coding pattern commonly used in the POET library to build a list of the necessary replacements and then apply all the replacements in the end.

```
  repl=NULL;
  foreach decl=TypeInfo#(type=_,_,_) \in input do
      repl=BuildList( (type, TranslateCtype2Ftype(type)), repl);
  enddo
  input = REPLACE(repl, input);
```

Here to translate variable variable declarations in C syntax to those in Fortran, the above code first builds an empty replacement list and uses the variable *repl* to keep track of this list. It then goes over the entire input in reverse order and locates all the $VarTypeDecl$ objects. The code then extends the *repl* list by pre-pending a new replacement (a (source, dest) pair) specification to it. After traversing the entire input, the list of replacements in *repl* is then applied collectively by invoking the $REPLACE$ operation on *input*.

11 Other Tasks

Value Numbering. As value number is a combined process of hash table management and AST modification, it is obvious that the POET associative map should be used to implement the hashing of value numbers and variable names, and that AST modification should be implemented by invoking the REPLACE operation.