Adding Equations to PCCTS

Lutz H. Hamel
Programming Research Group
Oxford University Computing Laboratory
e-mail: Lutz.Hamel@comlab.ox.ac.uk

Abstract

PCCTS is a highly integrated compiler construction tool. It supports attribute grammar style specifications of compilers, as well as the more traditional semantic actions. Its EBNF production rule notation and the implicit and explicit parse tree generation facilities, amongst many other features, provide additional support to the compiler writer. However, the use of equations for the specification of compilers has not received much attention, even though equations provide a powerful specification mechanism due to their abstract nature and their simple semantics. With the recent advances in term rewriting technology (rates of up 500,000 rewrites per sec.) we argue that equations provide a viable compiler specification alternative and should be integrated into the PCCTS tool. Using the UCG-E equational specification language as a basis we show how equational specifications could be integrated into PCCTS and study an interpreter for a small imperative language.

1 Introduction

PCCTS is a highly integrated compiler construction tool. It supports attribute grammar style specifications of compilers, as well as the more traditional semantic actions. Its EBNF production rule notation and the implicit and explicit parse tree generation facilities, amongst many other features, provide additional support to the compiler writer.

However, it was recognized in the late 70's and early 80's that first-order equations provide an expressive formalism for the specification of programming languages; in particular with respect to the specification of programming languages semantics [2, 7, 12]. More recently we were able to show that efficient, industrial strength compilers for languages as complicated as C++ can also be specified equationally [4]. We argue that the attraction of equations as a specification formalism is due to their abstract nature and simple semantics: *replacing equals with equals*. This simplicity results in very clean and intuitive specifications which are easy to understand, maintain and extend. Furthermore, equations can easily and efficiently be implemented (executed) on a computer by viewing them as rewrite rules from left to right; in other words, by viewing them as specialized equations which only work from left to right. Given the recent advances in term rewriting technology (rates of up 500,000 rewrites per sec. [9]), equational reasoning can be performed very fast and thus equational specification should be considered a viable alternative for the specification of compilers.

To demonstrate how equations can be used for language and compiler specifications we built an interpreter for a small but sufficiently complex imperative language which not only supports C-like language constructs such as `if` and `while` statements but also function calls and simple I/O. The syntax of the language is specified using the PCCTS system [11]. Its semantics is specified with the UCG-E equational specification language [5]. Since the equations of the UCG-E specification language are executable, the specification of the semantics of the language constitutes an executable interpreter for the language.

Finally, we present some ideas of how equational specifications could be incorporated into PCCTS specification file, providing a highly integrated compiler construction environment which allows the compiler writer to mix and match representation and specification styles as needed for a particular language implementation.

The rest of the paper proceeds as follows. We first take a look at equations and equational specifications *per se* and try to motivate their use in Sect. 2. We then briefly discuss the UCG-E specification language in Sect. 3. Sect. 4 introduces our programming language. We briefly look at the parser specification for the language in Sect. 5. The equational specification of the interpreter is discussed in Sect. 6. Some thoughts and comments of how equations could be incorporated into the existing PCCTS system are presented in Sect. 7 and we finish with some final remarks in Sect. 8.

2 Why Equations?

We were attracted to equations as a specification language due to their declarative nature and their simple semantics,[1] in particular when viewed as directed rewrite rules [8]: a term which matches the left side of an equation is simply replaced by the term on the right side of that equation. The rewrite process continues until no further matches can be identified.

Formalizing this a little bit; an equational specification consists of a signature Σ, a set X of variable names and a set E of equations. The signature defines the available operation symbols in the specification. Each equation in E has the form

$$l = r$$

where l and r are terms over the signature and a suitable set of variables. Sometimes we write $l, r \in T_\Sigma(X)$ where $T_\Sigma(X)$ is the set of all possible terms over a signature Σ and a set X of variable names. The execution of an equational specification consists of exhaustively applying the equations in E as rewrite rules to an appropriate input term. Any subterm (including previously rewritten terms) which matches the left side of an equation is replaced by the right side of the equation taking the assignments to the variables into account. As an example consider the following equational specification which specifies the `append` of an item to a list of items:

```
SIGNATURE
  nullary:  1, 2, 3, nil
  binary:   append, cons
  var:      ITEM, HEAD, TAIL

EQUATIONS
  append(ITEM,nil) = cons(ITEM,nil)
  append(ITEM,cons(HEAD,TAIL)) = cons(HEAD,append(ITEM,TAIL))
```

Here the signature, the variable set and the set of all possible terms over the signature and the variables are:

$$\begin{aligned} \Sigma &= \{\texttt{1}, \texttt{2}, \texttt{3}, \texttt{nil}, \texttt{append}, \texttt{cons}\}, \\ X &= \{\texttt{ITEM}, \texttt{HEAD}, \texttt{TAIL}\}, \\ T_\Sigma(X) &= \{\texttt{1}, \texttt{nil}, \texttt{append}(\texttt{ITEM}, \texttt{nil}), \texttt{cons}(\texttt{1}, \texttt{nil}), \ldots\}, \end{aligned}$$

respectively. It is easy to see that the left sides and the right sides of the equations are elements in $T_\Sigma(X)$. Now, given the input term `append(3,cons(1,cons(2,nil)))`, the exhaustive application of the equations in the specification above yields the term `cons(1,cons(2,cons(3,nil)))` by first applying the second equation twice and then the first equation once. Here is the trace of the computation:

```
append(3,cons(1,cons(2,nil)))
   { apply second equation }
cons(1,append(3,cons(2,nil)))
   { apply second equation }
cons(1,cons(2,append(3,nil)))
   { apply first equation }
cons(1,cons(2,cons(3,nil)))
```

[1] For aficionados: the denotational semantics of equations is given by general algebra [1] and their operational semantics is given by term rewriting [10]

The last term in the trace is also called the *normal form*, since there are no more equations to further *reduce* the term.

Another reason that attracted us to the use of equations is the fact that equations and equational rewriting match very well the intuitive view a programmer has of the compilation process: a successive rewriting of the abstract syntax tree (as generated by the parser) into an intermediate form – the "semantic" tree, which represents the program after the necessary coherence and type checks have been performed. Consider the following equational specification of a C-like type checker; type information is propagated throughout the semantic tree and operations are promoted as is deemed necessary.

```
SIGNATURE
  unary:    int, float, itof
  binary:   plus, intplus, floatplus
  var:      L, R

EQUATIONS
  plus(int(L),int(R)) = int(intplus(L,R))
  plus(float(L),int(R)) = float(floatplus(L,itof(R)))
  plus(int(L),float(R)) = float(floatplus(itof(L),R))
  plus(float(L),float(R)) = float(floatplus(L,R))
```

The operation symbols `int` and `float` represent the types of the corresponding subtrees. The symbol `itof` represents a type promotion operation from integer to floating point. The symbol `plus` represents a generic plus operation which the type checker promotes to the appropriate plus operation such as `intplus` or `floatplus`.

Let us examine the equations a little bit closer. The first equation states that a `plus` node with two integer subtrees itself becomes an integer subtree with the addition being an integer addition. Analogously, the fourth equation states that a `plus` node with two float subtrees is itself a float subtree with a floating point addition. On the other hand, the second and third equations state that a `plus` node with an integer subtree as well as a float subtree turns into a float subtree with the addition being a floating point addition and its integer subtree explicitly promoted to a float subtree. It is precisely this terseness of the specification of rather complex problems that makes equations attractive for the construction of compilers and interpreters.

3 The UCG-E Specification Language

The UCG-E specification language [5] is a descendant of the UCG system [6] developed at the University of New Hampshire for the generation of code generators. The UCG-E system takes an equational specification and translates it into C++ code. The generated code consists of a lookup table and a set of C++ function definitions. Each specification has two parts: a *declaration section* which includes the signature and variable declarations and an equational or *rule section*. The two sections are separated by a **%%** symbol. Each equation in the rule section has the form

$$l := r$$

where $l, r \in T_\Sigma(X)$ and represents a directed rewrite rule, given an appropriate signature Σ and a variable set X. Two restrictions apply to the form of these rewrite rules:

- The term l must be linear, i.e., each variable in l may appear only once,
- $var(r) \subseteq var(l)$, where var is a function which given a term in $T_\Sigma(X)$ returns the set of variables occurring in that term.

The first restriction insures that if a rule matches an input term the variables in l are uniquely instantiated. The second restriction insures that instead of having to do a global unification of the variables we may simply copy the values of the variables from the left side to the right side during rewriting if necessary.

As a concrete example, here is the type checker example from the previous section written in the UCG-E specification language:

```
/* SIGNATURE */
%unary int;
%unary float;
%unary itof;
%binary plus;
%binary intplus;
%binary floatplus;
%var TERM L;
%var TERM R;

%%

/* EQUATIONS */
plus(int(L),int(R)) := int(intplus(L,R));
plus(float(L),int(R)) := float(floatplus(L,itof(R)));
plus(int(L),float(R)) := float(floatplus(itof(L),R));
plus(float(L),float(R)) := float(floatplus(L,R));
```

Two features make UCG-E especially attractive for the use in industrial software production. These are the *term generating functions* and the *user action functions*. Both of these features essentially provide an interface from UCG-E to software systems written in C++.

UCG-E builds a term generating function for each symbol in the signature. The user may call these functions to build terms labeled by symbols from the signature. In addition to generating a term these functions make the new term known to the UCG-E term rewriting mechanism which attempts to apply any of the given rules to this newly constructed term. The name for a term generating function is derived from a symbol in the signature by capitalizing its name and sticking a `GEN_` in front of it. For example, the term generating function for the symbol `int` in the above specification is `GEN_INT`.

The user action functions, on the other hand, allow the use of side effects in the rewrite rules. These side effects could take on the form of I/O, a symbol table mechanism, or any other action which lies outside the realm of efficient equational processing. User action functions are distinguished symbols in the signature of the specification and may only appear on the right hand side of the rewrite rules. The keyword `%func` flags a symbol in the signature as a user action function.

Another feature which is important and quite powerful is the fact that one can embed C++ types in UCG-E's term structure via *attributes*. These types can be pointers, special constants or just values which cannot be efficiently represented with UCG-E's term structure. Again, the idea being a frictionless communication between the C++ environment and the equational specification system. Consider the following which specifies how to append a C++ integer to a list of C++ integers:

```
/* SIGNATURE */
%nullary nil;
%nullary item [int i];   /* `item´ has `int i´ as an attribute */
%binary  cons;
%binary  append;
%var TERM HEAD;
%var TERM TAIL;
%var int VAL;

%%

/* EQUATIONS */
append(item(VAL),nil) := cons(item(VAL),nil);
append(item(VAL),cons(HEAD,TAIL)) := cons(HEAD,append(item(VAL),TAIL));
```

Please note that `item` carries C++ integers as an attribute and UCG-E allows us to do rewriting on attributes as if these were part of the actual term structure.

In general, term rewriting systems worry not only about the final answer, but also that the same answer is obtained via any competing rewrite sequences given the same input term. This is known as *confluency*. Rather than imposing any further restrictions on the form of the rules, UCG-E assumes the specification to be confluent. Since UCG-E allows side effects to be used in its rewrite rules, the rewrite sequence is as much part of the solution as is the final answer and therefore a deterministic selection of the rewrite sequence is important. UCG-E selects rules by the order in which they appear in the specification. It guarantees that rules appearing earlier in the specification have a higher priority than rules appearing later in the specification.

The programming and debugging of equational specifications is greatly facilitated by the interactive graphical debugger supported by UCG-E. The debugger allows the user to single-step rewrite sequences one rule at a time and to browse current state information.

4 The TL Language

To illustrate some of the power and elegance of equational specifications we constructed an interpreter for the simple imperative language TL (as in **T**oy **L**anguage). The language supports some of the "standard" language constructs such as `if` and `while` statements, but in order to make things a little bit more interesting we also included function calls and simple I/O. However, to keep the complexity somewhat manageable we restricted ourselves to the support of integral types. To convey the flavor of the language, consider the following program written in TL which computes the factorial of `i`:

```
get i;
k = 1;
fact = 1;

while (k <= i)
{
        fact = fact * k;
        k = k + 1;
}

put fact;
```

In a TL program execution always starts at the first statement in the file – here the `get i` statement which reads a value for `i` from the input. TL does not have declaration statements but rather the first mention of a name declares that name (we only have integer types). We also do not have scoping, we only have a flat global name space. For people familiar with C the above program should not hold any surprises and should be fairly self-explanatory.

5 The TL Parser Specification

Before looking at the parser specification for TL it is worth mentioning that UCG-E has one built-in type, namely `TERM`. Any term which is constructed via UCG-E's term generation functions is of this type. In order to get a better feel of how exactly the communication between the parser and UCG-E is handled, let us look at the piece of parser specification which tells us how statements are parsed in TL (we assume familiarity with the PCCTS system [11]):

```
statement >[TERM t] :      << TERM lv; TERM e; TERM st; TERM sl; >>
          lval >[lv] "=" expression >[e] ";"
                << $t = GEN_ASSIGN(lv,e); >>
        | GET lval >[lv]  ";"
                << $t = GEN_GET(lv); >>
        | PUT lval >[lv] ";"
```

```
                << $t = GEN_PUT(lv); >>
        | IF "\(" expression >[e] "\)" statement >[st]
                << $t = GEN_IF(e,st); >>

                        ...
        ;
```

The first line in this specification says that the non-terminal `statement` returns a value of type `TERM`, i.e., a UCG-E term. Next, between the `<< ...>>` brackets, we declare a couple of local variables for the use in the subsequent production rules. Looking at the first production rule we find that the result of parsing an `lval` is assigned to the local variable `lv` and that the result of parsing an `expression` is assigned to the variable `e`. Both are of type `TERM`. Once these subparts of an assignment statement have been recognized we construct an UCG-E assignment term via the `GEN_ASSIGN` term generating function. The arguments to the term generating function are the terms assigned to `lval` and `e`. The constructed term is returned as the result of the non-terminal `statement`. Similarly for the other production rules.

Even though this sounds somewhat complicated, the basic idea is to construct a term from the bottom up which represents the parsed syntax. This is very much like the explicit parse tree construction mechanism provided by PCCTS. The interesting part here is that every time one constructs a term with a term generating function, UCG-E looks into its internal rewrite table and sees if any rules match the current term. If so the term is immediately rewritten according to the rules found in the table. A complete specification of the TL parser is given in Appendix A.

6 The Equational Specification of the TL Interpreter

Up to this point we have only looked at the syntax of TL. We have looked very informally at TL programs but we have not said anything concrete about the meaning of the syntactic elements of the language or how they should be interpreted. One way to assign meaning to a syntactic construct is to have an evaluation function assign a value to that construct. In this case, interpreting a syntactic construct is nothing else than computing its value. Given this, the meaning of a program is a composition of the values of its constituent parts, i.e., the values of its statements, expressions, *etc.* Thus, in order to interpret a program we have to first compute the values of its constituent parts and then the value of their composition. This is precisely the approach we take here. [2]

As we have seen, equations are powerful and allow one to express fairly complicated computations quite elegantly. We demonstrate this with the equational specification of the evaluation function `eval` which computes the meaning of each syntactic construct. In our case the `eval` function computes a value `v` for each construct. Furthermore, since our equations are executable, the equational specification of the `eval` function constitutes an executable interpreter for TL.

To drive some of these points home, let us take a look at the equational specification of the evaluation of a couple of constructs. In what follows, `v` represents a value which has an attribute of type `int` and names in capital letters represent variables of "just the right type". Let us start with something simple, for example the `get` statement:

```
eval(get(id(SYM))) := v(input_val(SYM));
```

This equation states that the meaning of a `get` statement is the input value for the identifier `SYM`. Here, `input_val` is a user action function which reads a value from the terminal, assigns it to the identifier `SYM` and returns the value as the attribute for `v`. Similarly for the `put` statement:

```
eval(put(id(SYM))) := v(output_val(SYM));
```

As above, this equation states that the meaning of a `put` statement is the output value of the identifier `SYM`. Let us look at something a little more exciting – the assignment statement:

[2] Our approach is very similar to the "denotational" semantics developed by Strachey and Scott in the early 70's [3] but differs substantially in its mathematical foundations – no domain theory!

```
eval(assign(id(SYM),E)) := a(id(SYM),eval(E));
a(id(SYM),v(VAL)) := v(assoc_val(SYM,VAL));
```

Since the assignment statement is a little more complicated than the constructs considered so far, we make use of an auxiliary function `a` (as in **a**ssignment). The first equation says that the meaning of the assignment of an expression `E` to an identifier `SYM` is computed by the application of the function `a` to the identifier `SYM` and the evaluation of the expression `E`. The next equation tells us exactly how the application of `a` is to be computed once the evaluation of `E` is complete: associate the value computed for `E` with the identifier `SYM`. Furthermore, because equations are transitive we may also say that the meaning of an assignment is the value of the association of the expression on its right side with the identifier on its left side. Let us look at one more construct – the `if` statement:

```
eval(if(COND,STMT)) := c(eval(COND),STMT);
c(v(FALSE),STMT) := v(NULL);
c(v(!FALSE),STMT) := eval(STMT);
```

The first equation tells us that the meaning of an `if` statement is the application of the auxiliary function `c` (as in **c**onditional) to the evaluation of the condition `COND` and the statement `STMT`. Similar to the case of the assignment statement, the next two equations tell us how the application of `c` is to be computed once the evaluation of the condition is completed: if the evaluation of the condition returns a value `FALSE` we just return a `NULL` value for the meaning of the `if` statement, otherwise we return the value due to the evaluation of the `STMT`.

The question one might ask at this point is – how does the evaluation process get started? To answer this we need to take another look at the parser specification. There we find the following production:

```
program :
          statement_list >[TERM sl] Eof
                << GEN_EVAL(term = sl); >>
        ;
```

This production says that a program is a statement list followed by an end-of-file marker. As we can see form the production, parsing a statement list produces a term assigned to `sl`. Now, in order to compute the meaning of a program we simply need to compute the meaning of the statement list that makes up the program by evaluating the term `sl`. This is accomplished by contructing an `eval` term which has the term assigned to `sl` as an argument. As soon as the term is constructed any rules which match the term will be applied, thus starting the evaluation of the entire program. It might be worthwhile at this point to take a closer look at the full equational specification of the TL language in Appendix B

We hope that the above examples have adequately illustrated the fact that equations provide a powerful tool for the specification of language processing software. It should be clear that the techniques shown here can easily be extended to the specification of compilers – rather than specifying the meaning of a language construct in terms of a value, one could specify its meaning in terms of code to be generated or actions to be performed on a symbol table.

7 Equations in the PCCTS Environment

PCCTS is a highly integrated compiler construction tool the strength of which lies precisely in the fact that it allows the programmer to mix and match the representation and specification styles needed for a particular language implementation. Since we argue that equations are another alternative for the specification of language processing software, we present some ideas here of how equations could be integrated into the PCCTS environment. Consider for a moment the piece of PCCTS grammar from above which specifies TL statements. It would be nice to have syntactic properties and semantic properties of the language specified close together so that when changes are being made coherence of the specification is more readily maintained. Our idea is to intersperse syntactic production rules with equations defining the semantics of the TL constructs. For example:

```
statement >[TERM t] :      << TERM lv; TERM e; TERM st; TERM sl; >>
          lval >[lv] "=" expression >[e] ";"
                << $t = #assign(lv,e); >>
#eq eval(assign(id(SYM),E)) := a(id(SYM),eval(E))
#eq a(id(SYM),v(VAL)) := v(assoc_val(SYM,VAL))
        | GET lval >[lv]  ";"
                << $t = #get(lv); >>
#eq eval(get(id(SYM))) := v(input_val(SYM))
        | PUT lval >[lv] ";"
                << $t = #put(lv); >>
#eq eval(put(id(SYM))) := v(output_val(SYM))
        | IF "\(" expression >[e] "\)" statement >[st]
                << $t = #if(e,st); >>
#eq eval(if(COND,STMT)) := c(eval(COND),STMT)
#eq c(v(FALSE),STMT) := v(NULL)
#eq c(v(!FALSE),STMT) := eval(STMT)

                    ...
        ;
```

Here we chose a syntax for the term constructors appearing in the productions very similar to the syntax of the explicit parse tree constructors already supported by PCCTS; i.e., `#assign(lv,e)` builds an assignment term whose left subterm is `lv` and whose right subterm is `e`. Equations are preceded by the keyword `#eq`. As in the case of UCG-E, the constructed term is immediately rewritten if the left sides of any of the equations found in the specification match the term.

8 Conclusions

PCCTS is a highly integrated compiler construction tool the strength of which is due to the fact that it allows the compiler writer to mix and match representation and specification styles as needed for a particular language implementation. However, one important formalism is not supported by PCCTS – equational specifications.

We argue that the abstract nature and the simple semantics of equations make them an ideal specification formalism. In addition, equations can be easily implemented on a computer by viewing them as directed rewrite rules. Given the recent advances in term rewriting technology, we feel that equations present a real alternative to be considered for the specification of compilers and interpreters and should be integrated into the PCCTS system.

After introducing equations and equational specifications very informally and showing how to write equational specifications in the UCG-E language, we illustrated how equational specifications could be applied to language processing software by constructing an interpreter for a small language. We feel that the resulting specification speaks for itself in terms of clarity and maintainability.

Finally we presented some ideas of how to integrate equations into PCCTS. The idea being that it is important to have the specification of the syntax and the semantics of a language as close together as possible, since this will insure that coherence of the language specification will be maintained more readily. Overall, it seems to us that this high level of integration and the ability to mix various styles of specification will result in extremely intelligible and maintainable compilers and other language processors.

References

[1] Joseph Goguen. *Theorem Proving and Algebra.* MIT, 1994.

[2] Joseph Goguen and Kamran Parsaye-Ghomi. Algebraic denotational semantics using parameterized abstract modules. In J. Diaz and I. Ramos, editors, *Formalizing Programming Concepts*, pages 292–309. Springer, 1981. Lecture Notes in Computer Science, Volume 107.

[3] Michael J.C. Gordon. *The Denotational Description of Programming Languages.* Springer, 1979.

[4] Lutz H. Hamel. Industrial strength compiler construction with equations. *ACM SIGPLAN Notices*, 27(8), August 1992.

[5] Lutz H. Hamel. UCG-E: An equational logic programming system. In *Proceedings of the Programming Language Implementation and Logic Programming Symposium 1992*, Lecture Notes in Computer Science 631. Springer-Verlag, 1992.

[6] Philip J. Hatcher. The equational specification of efficient compiler code generation. *Comp. Lang.*, 16(1):81–96, 1991.

[7] Christoph Hoffmann and Michael O'Donnell. Programming with equations. *Transactions on Programming Languages and Systems*, 1(4):83–112, 1982.

[8] Gérard Huet and Derek Oppen. Equations and rewrite rules: A survey. In Ron Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 349–405. Academic, 1980.

[9] J.F.Th. Kampermann and H.R. Walters. ARM – Abstract Rewriting Machine. Technical report, CWI, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands, 1994. WWW: http://www.cwi.nl/ftp/gipe/index.html.

[10] Claude Kirchner, Hélène Kirchner, and José Meseguer. Operational semantics of OBJ3. In T. Lepistö and Aarturo Salomaa, editors, *Proceedings, 15th International Colloquium on Automata, Languages and Programming, Tampere, Finland, July 11-15, 1988*, pages 287–301. Springer, 1988. Lecture Notes in Computer Science, Volume 317.

[11] T.J. Parr, H.G. Dietz, and W.E. Cohen. PCCTS Reference Manual. WWW: ftp://ftp-mount.ee.umn.edu/pub/pccts, August 1991.

[12] Mitchell Wand. First-order identities as a defining language. *Acta Informatica*, 14:337–357, 1980. Originally Report 29, Computer Science Deptartment, Indiana University, 1977.

Notes

The systems mentioned in this paper are available on the Internet. The addresses given here are anonymous ftp addresses, we give them in the WWW URL format.

- PCCTS is available from the following sites:

  ftp://ftp-mount.ee.umn.edu/pub/pccts

  ftp://ftp.parr-research.com/pub/pccts

- UCG-E and the TL interpreter developed here are available from:

  ftp://ftp.comlab.ox.ac.uk/tmp/Lutz.Hamel/software

- The UCG-E manual is available from:

  ftp://ftp.comlab.ox.ac.uk/tmp/Lutz.Hamel/papers

A The PCCTS TL Parser Specification

```
/*
 *  The PCCTS grammar for TL.
 *
 *  Note: the GEN_xxx functions are provided by the UCG-E compiler.
 *
 *  (c) 1995 -- Lutz H. Hamel.
 */

#header <<
extern "C" {
#include <stdio.h>
#include <stdlib.h>
}
#include "interp.h"
#include "uudef.hxx"
>>

#token ASSIGN "="
#token OR "\|\|"
#token AND "&&"
#token EQUAL "=="
#token NOTEQUAL "!="
#token LESSTHAN "<"
#token GREATERTHAN ">"
#token LESSEQUAL "<="
#token GREATEREQUAL ">="
#token PLUS "\+"
#token MINUS "\-"
#token MULT "\*"
#token DIVIDE "/"
#token NOT "!"

#token IF "if"
#token WHILE "while"
#token GET "get"
#token PUT "put"
#token FUNC "func"
#token RETURN "return"
#token CALL "call"

#token ID "[a-zA-Z_][a-zA-Z0-9_]*"
#token CONST "([1-9][0-9]*)|0"

#token "/\*" << mode (COMMENT); skip (); >>

#token "[\t\ ]+"                          << skip (); >>
#token "[\n\r]"                           << newline(); skip(); >>
#token "// ~[\n]* \n"                     << newline(); skip(); >>

#token Eof "@"

#lexclass COMMENT
```

```
#token "[\n\r]"            << skip(); newline(); >>
#token "\*/"               << mode (START); skip (); >>
#token "\*~[/]"            << skip (); >>
#token "~[\*\n\r]+"        << skip (); >>

#lexclass START

<<
typedef ANTLRCommonToken ANTLRToken;

extern TERM term;
>>


class TLParser
{
<<
public:
        void init()             { ANTLRParser::init(); }
        // if you want a trace -- comment the following two function out.
        void tracein(char *r)   {;}
        void traceout(char *r)  {;}
>>



program :
          statement_list >[TERM sl] Eof
                << GEN_EVAL(term = sl); >>
        ;

/* ugly -- but it allows us to construct nicer terms with UCG-E */
statement_list >[TERM t] :
          statement >[TERM s] statement_list >[TERM sl]
                << $t = GEN_STMT_LIST(s,sl); >>
        | /* empty */
                << $t = GEN_NIL(); >>
        ;

statement >[TERM t] :             << TERM lv; TERM e; TERM st; TERM sl; >>
          lval >[lv] ASSIGN expression >[e] ";"
                << $t = GEN_ASSIGN(lv,e); >>
        | GET lval >[lv]  ";"
                << $t = GEN_GET(lv); >>
        | PUT lval >[lv] ";"
                << $t = GEN_PUT(lv); >>
        | "\{" statement_list >[sl] "\}"
                << $t = GEN_BLOCK(sl); >>
        | IF "\(" expression >[e] "\)" statement >[st]
                << $t = GEN_IF(e,st); >>
        | WHILE "\(" expression >[e] "\)" statement >[st]
                << $t = GEN_WHILE(e,st); >>
        | FUNC id1:ID "\(" "\)" "\{" statement_list >[sl] "\}"
                << $t = GEN_FUNCDEF(table->InstallSym($id1->getText()),sl); >>
        | RETURN expression >[e] ";"
```

```
                << $t = GEN_RETURN(e); >>
         | CALL id2:ID "\(" "\)" ";"
                << $t = GEN_FUNCREF(table->InstallSym($id2->getText())); >>
        ;

expression >[TERM t] :
          logical_or_expression >[$t]
        ;

logical_or_expression >[TERM e] :
          logical_and_expression >[$e]
          ( OR logical_and_expression >[TERM e1]
                << $e = GEN_BINOP(OR,$e,e1); >> )*
        ;

logical_and_expression >[TERM e] :
          equality_expression >[$e]
          ( AND equality_expression >[TERM e1]
                << $e = GEN_BINOP(AND,$e,e1); >> )*
        ;

equality_expression >[TERM e] :
          relational_expression >[$e]
                << OP op = LA(1); >>
          ( (NOTEQUAL | EQUAL) relational_expression >[TERM e1]
                << $e = GEN_BINOP(op,$e,e1); >> )*
        ;

relational_expression >[TERM e] :
          additive_expression >[$e]
                << OP op = LA(1); >>
          ( (LESSTHAN | GREATERTHAN | LESSEQUAL | GREATEREQUAL)
            additive_expression >[TERM e1]
                << $e = GEN_BINOP(op,$e,e1); >> )*
        ;

additive_expression >[TERM e] :
          multiplicative_expression >[$e]
                << OP op = LA(1); >>
          #pragma approx
          ((PLUS | MINUS) multiplicative_expression >[TERM e1]
                << $e = GEN_BINOP(op,$e,e1); >> )*
        ;

multiplicative_expression >[TERM e] :
          unary_expression >[$e]
                << OP op = LA(1); >>
          ((MULT | DIVIDE) unary_expression >[TERM e1]
                << $e = GEN_BINOP(op,$e,e1); >> )*
        ;

unary_expression >[TERM e] : << TERM e1; >>
          MINUS expression >[e1]
                << $e = GEN_UOP(MINUS,e1); >>
        | NOT expression >[e1]
```

```
                << $e = GEN_UOP(NOT,e1); >>
        | primary_expression >[$e]
        ;

primary_expression >[TERM e] :
           id1:ID
                << $e = GEN_ID(table->InstallSym($id1->getText())); >>
        |  id2:ID "\(" "\)"
                << $e = GEN_FUNCREF(table->InstallSym($id2->getText())); >>
        |  con:CONST
                << $e = GEN_CON(atoi($con->getText())); >>
        |  "\(" expression >[$e] "\)"
        ;

lval >[TERM e] :
         id:ID
                << $e = GEN_ID(table->InstallSym($id->getText())); >>
        ;
```

B The Equational Specification of the TL Semantics

```
/*
 *         TL -- equational specification of the semantics.
 *
 *        (c) 1995 -- Lutz H. Hamel.
 */

/* Declaration section. */

%include "tokens.h";
%include "interp.h";
%include "symtab.h";

/* tell UDB how to display data types */
%view view_p_sym(P_SYM p);
%view view_int(int i);
%view view_op(OP op);

/* SIGNATURE */

/* term constructors */
%nullary id         [P_SYM psym];
%nullary con        [int val];
%nullary nil;
%unary get;
%unary put;
%unary uop          [OP op];
%unary block;
%unary return;
%binary binop       [OP op];
%binary assign;
%binary stmt_list;
%binary if;
%binary while;
%unary   funcdef    [P_SYM name];
%nullary funcref    [P_SYM name];

/* interpretation functions */
%unary eval;
%binary a;                          /* assignments */
%binary b           [OP oper];      /* binary operators */
%unary u            [OP oper];      /* unary operators */
%binary c;                          /* conditionals (if) */
%binary lc;                         /* loop condition */
%binary ls;                         /* loop statement */
%unary r;                           /* return statement */
%binary s;                          /* statement */
%unary find_func    [P_SYM name];

/* value constructor */
%nullary v          [int val];

/* declare variables used in the equations. */
%var TERM T;
```

```
%var TERM T1;
%var TERM T2;
%var TERM COND;
%var TERM STMT;
%var TERM LST;
%var TERM E;
%var TERM BL;
%var P_SYM SYM;
%var P_SYM NAME;
%var P_SYM NAME1;
%var P_SYM NAME2;
%var int VAL;
%var int VAL1;
%var int VAL2;
%var OP OPER;

/* declare constants used in the equations. */
%const FALSE;
%const NULL;

/* declare user action functions. */
%func int input_val(P_SYM psym);
%func int output_val(P_SYM psym);
%func int assoc_val(P_SYM psym, int val);
%func int get_val(P_SYM psym);
%func int fold_val(OP op, int val1, int val2);
%func int find_and_eval(P_SYM name);
%func TERM is_func(P_SYM name1, P_SYM name2, TERM func_def, TERM list);
%func int get_return_val(TERM block);
%func int make_return_val(int val);
%func TERM error(STRING str);

%%

/* EQUATIONS */

/* interpret a ´read´ node. */
eval(get(id(SYM))) := v(input_val(SYM));

/* interpret a ´write´ node. */
eval(put(id(SYM))) := v(output_val(SYM));

/* interpret an ´assign´ node. */
eval(assign(id(SYM),E)) := a(id(SYM),eval(E));
a(id(SYM),v(VAL)) := v(assoc_val(SYM,VAL));

/* interpret a ´block´. */
eval(block(T)) := eval(T);

/* interpret a ´con´. */
eval(con(VAL)) := v(VAL);

/* interpret a ´sym´. */
eval(id(SYM)) := v(get_val(SYM));
```

```
/* interpret a ´binop´. */
eval(binop(OPER,T1,T2)) := b(OPER,eval(T1),eval(T2));
b(OPER,v(VAL1),v(VAL2)) := v(fold_val(OPER,VAL1,VAL2));

/* interpret a ´uop´. */
eval(uop(OPER,T)) := u(OPER,eval(T));
u(OPER,v(VAL)) := v(fold_val(OPER,NULL,VAL));

/* interpret an ´if´ statement. */
eval(if(COND,STMT)) := c(eval(COND),STMT);
c(v(FALSE),STMT) := v(NULL);
c(v(!FALSE),STMT) := eval(STMT);

/* interpret a ´while´ statement. */
eval(while(COND,STMT)) := lc(eval(COND),while(COND,STMT));
lc(v(FALSE),while(COND,STMT)) := v(NULL);
lc(v(!FALSE),while(COND,STMT)) := ls(eval(STMT),while(COND,STMT));
ls(T,while(COND,STMT)) := eval(while(COND,STMT));

/* interpret a ´function reference´ */
eval(funcref(NAME)) := v(find_and_eval(NAME));

/* interpret a ´function definition´ */
eval(funcdef(NAME,BL)) := v(get_return_val(eval(BL)));

/* interpret a ´return´ */
eval(return(E)) := r(eval(E));
r(v(VAL)) := v(make_return_val(VAL));

/* describe how to deal with statement lists. */
/* NOTE: the first two equations are special cases -- special attention */
eval(stmt_list(funcdef(NAME,BL),LST)) := eval(LST);
eval(stmt_list(return(E),LST)) := eval(return(E));
/* NOTE: here we make use of the rhs evaluation sequence enforced by UCG:
   ´eval(STMT)´ is evaluated before ´eval(LST)´, which is exactly
   what we want. */
eval(stmt_list(STMT,LST)) := s(eval(LST),eval(STMT));
s(v(NULL),v(NULL)) := v(NULL);
eval(nil) := v(NULL);

/* find a function definition */
find_func(NAME1,stmt_list(STMT:funcdef(NAME2,BL),LST))
         := is_func(NAME1,NAME2,STMT,LST);
find_func(NAME,stmt_list(STMT,LST)) := find_func(NAME,LST);
find_func(NAME,nil) := error("no such function");
```