EULER
An Experiment in Language Definition

Thomas W. Christopher

Department of Computer Science
and Applied Mathematics

Illinois Institute of Technology

The EULER compiler itself is public domain. Since it is in the public domain, it may be copied and used without restriction. The author makes no warranties of any kind as to the correctness of EULER or its suitability for any application. The responsibility for the use of the program lies entirely with the user.

To contact the author

Thomas Christopher
Department of Computer Science and Applied Mathematics
Illinois Institute of Technology
IIT Center
Chicago IL 60616 USA

tc@charlie.cns.iit.edu
http://www.iit.edu/~tc

To obtain a up-to-date copy of this document, EULER, and the TCLL1 parser generator

http://www.iit.edu/~tc/toolsfor.htm

Release date: 3/22/97

Acknowledgement:

My thanks to Patricia Guilbeault for technical editing.

Contents

List of Tables

Chapter 1 An Experiment in Language Definition

It was the mid-1960's, and ALGOL 60 was a success, or at least, the ALGOL 60 Revised Report[1] was a success. The language itself caught on mainly in Europe, leaving FORTRAN to the Americans. But the Report established standards for the definition of programming languages. It introduced BNF (Backus Normal Form or Backus-Naur Form) as a way of writing context-free grammars for programming languages. It defined the semantics of ALGOL 60 by using English language text to describe the meanings of the productions. The meaning of a program could be understood as a composition of the meanings of the phrases of which it was composed. At least that was the theory.

There were problems, though. The ALGOL 60 grammar was not fully reduced: it had nonterminals and productions solely for discussion but which could not occur in actual derivations. Even with a clean grammar, compiler writers would not have had an easy time. Efficient parsing techniques for context-free grammars had not yet been developed. The semantic descriptions were not totally clear and their implications were not fully understood. Compiler-writing was still a new discipline. ALGOL 60 had language features that compiler writers did not yet know how to implement. ALGOL 60's problems inspired half a decade of research.

For the most part, the research was successful. Now we define programming languages using context-free grammars for which we have a number of linear time parsing algorithms. We know how to implement block structure, how to pass local procedures as parameters, and how to implement ALGOL 60's notorious call-by-name—and we know better than to equip programming languages with call-by-name.

But what is still not clear is how we should specify the semantics of a programming language. English text lacks the precision of mathematics, and researchers envied mathematicians' ability to write precise definitions and prove theorems. Perhaps there is some mathematical way to define the semantics of programming languages—or if not exactly mathematical, then at least some concise, formal notation.

The ALGOL 68 Report attempted a more formal definition. The language ALGOL 68 was being designed at approximately the same time as EULER using von Wijngaarden's notation. One of the factors contributing to the failure of

---

[1] Naur, P. (ed.) Revised report on the algorithmic language ALGOL 60. *Comm. ACM* 6 (Jan. 1963).

ALGOL 68 is that its report was published before any informal introduction. Programmers looked at the ALGOL 68 Report and found such explanations as:

> { In rule a, `'ROWS'` reflects the number of trimscripts in the slice, `'ROWSETY'` the number of those which are trimmers and `'ROWWSETY'` the number of `'row of'` not involved in the indexer. In the slices *x2[i,j]*, *x2[i,2:n]*, *x2[i]*, these numbers are ***(2,0,0)***, ***(2,1,0)*** and ***(1,0,1)*** respectively. Because of rules **f** and 7.1.1.u, *2:3@0*, *2:n*, *2:*, *:5* and *:@0* are trimmers.}[2]

Programmers didn't understand the Report and gave up on the language. The attempt at a more formal definition worked against the success of the language.

Wirth and Weber attempted to create a formal method of defining programming languages in their paper "EULER: A Generalization of ALGOL, and its Formal Definition: Part I," and "—Part II"[3]. They contrasted their approach to those who were translating programs into the (calculus and to von Wijngaarden who was working on a system that would be used in the definition of ALGOL 68.

Wirth and Weber pointed out that one language "can only be explained in terms of another language which is already well understood." They found fault with the attempts to define programming languages in any terms other than programming. After all, if the point of a programming language is to communicate to a machine, what could be a more appropriate definition than one utilizing "elementary machine operations."

Their approach is to define a language by its compiler, but not simply providing a compiler as a black box upon which to perform experiments. The source code of the compiler is provided for inspection to aid understanding.

They introduced the ALGOL-like programming language EULER and tested their approach on it. They supplied semantics routines a compiler would execute during a parse of an EULER program. These routines manipulate a symbol table and place abstract machine instructions into an array. They supplied an interpreter for the abstract machine instructions.

Since they were defining the translator in terms of the actions the compiler takes during a parse of a program, they had to specify the parsing algorithm so that the order of actions would be completely clear. They devoted more than half of Part I of their paper to defining *simple precedence parsing* which they used for their compiler.

They assert that if one understands the language in which the translator and interpreter are written and the order of reductions performed by the parser, then one understands the meaning of an EULER program.

---

[2] von Wijngaarden, A. (ed.), Mailloux, B. J., Peck, J. E. L., and Koster, C. H. A., *Report on the Algorithmic Language ALGOL 68, Numerische Mathematik*, **14**, 79-218 (1969), page 168.

[3] *Communications of the ACM*, vol. 9, numbers 1 and 2, (Jan. and Feb. 1966).

Their approach has a number of clear advantages: It defines a programming language in terms a programmer is trained to understand. It proves it can be compiled. It makes it easier to port to a new machine.

Defining a language in terms of its compiler proves it can be compiled. ALGOL 60 had flaws in its designs that made compiling difficult. The ALGOL 60 designers apparently thought that they were specifying call-by-reference when they invented call-by-name. Dynamic **own** arrays require the implementer to provide some sort of heap allocation, although no other feature in the language can make use of a heap. Numeric statement labels complicate parameter passing—is this number an integer or a label? Or worse, is it a label that hasn't been declared yet? And can the subroutine use its parameter as both an integer and a label?

These problems became apparent when implementers attempted to compile ALGOL 60. If the language had been defined in terms of its compiler, then the problems would have been found by the language designers.

Wirth and Weber also point out that a language design based on a compiler would aid in language porting: a new compiler for a new machine can be seen to be correct by showing that the code generated is an "adaptation to particular environmental conditions of the language definition itself."

Defining a language by its compiler is not perfect, however. The compiler itself can have flaws, especially if it is only for reference and is not actually executable. Wirth and Weber had some slight flaws in their published EULER compiler:

- They are inconsistent in which field of an activation record is the static link and which is the dynamic link.
- They use an incorrect value for the static link when creating a procedural value.
- They need to push an initial activation record on the stack before running the program.
- They really should write out a "halt" instruction at the end of the program.

If you're going to define a language by its translation into another language, and you want the definition to be clear, then the target language should be at least as understandable as the source. The translator must itself be understandable. In the case of EULER, the target language is an abstract machine language—which is to say, the machine instructions for a fictitious computer. Some of the abstract machine operations are high level, complex instructions; you need to read the code of the interpreter to figure them out. So the interpreter needs to be understandable. Care must be paid to the coding practices, the algorithms, and the language they are written in.

Wirth and Weber wrote both the translator and the interpreter in EULER. EULER was a reasonable choice for the time, although it would no longer be preferred. EULER lacks records with named fields, and EULER lacks looping statements. Accessing fields by subscripting and coding loops with goto's both obscure their code. In fact, they themselves use a hidden representation of records—created and accessed through function calls. They added three hidden record types for references, program labels, and procedure closures. They give names for procedures to create records of these types and procedures to extract fields from the records. They require their type testing operations (**isr**, **isl**, and **isp**) to recognize these record types.

Using EULER for the translator and interpreter was a way to show off their technique for language definition, but it did leave some questions unanswered, such as:

- How does arithmetic work? You see that an EULER "+" operator is translated into a "+" abstract machine instruction, which is interpreted by an EULER "+" operator.
- How does subscripting work? It is defined in the interpreter by subscripting.
- How and where are lists allocated? Lists are created in the interpreter by EULER operations that create lists. It is not explicitly stated that the language needs a garbage collector, but it does need one.

Of course, if they had tried using a different implementation language, then they would have had other problems. There really weren't many good candidates at the time. Neither FORTRAN nor ALGOL 60 had the necessary data structures. Assembly language would have been too particular, verbose, and obscure. LISP would probably have been the best choice.

As we redo their work, we have the same problem. If we use a low level language such as C, our translator, interpreter, and run time system might be much longer and considerably more obscure than theirs. Besides, C is not cleanly defined itself. Instead, we use a very high level language, Icon. This opens us to the criticism that we are defining a simple language in terms of a more complex one. Moreover, Icon's semantics are not carefully defined. There is publicly available source code for Icon which can be consulted, but it is not intended as a definition. And the source code for Icon is written in C, bringing us back to the criticisms of C again.

Beyond specific problems with Wirth and Weber's attempt, there is a more general problem with defining a language in terms of its compiler: compilers may specify too much. For example, the code generated by the EULER compiler executes statements and expressions strictly left-to-right. What about a language where the execution order is not supposed to be specified? The compiler will pick a particular evaluation order. If the compiler is the definition of the language, then presumably the evaluation order must be the one chosen by the compiler. If the compiler tries not to specify the order, say by choosing randomly, then is a random choice the standard? And what about the error recovery? Are the error messages of any implementation required to be the same as the definition? Must the error recovery be the same?

We keep inventing complex languages. The compiler for a complex language is itself complex. Neither the compiler nor any other document will make understanding simple. Any complex definition will have bugs, inconsistencies, gaps, and failure to meet intentions.

Perhaps the best we can do is have multiple definitions:

- An informal description of the language with a grammar and an accompanying semantic description in English.
- A compiler.
- Suites of test programs that exercise all features of the language.

Each can aid in understanding the others, and the conflicts between them can bring the bugs in the definition to light.

To see the efficacy of defining a language in terms of its compiler, we redo Wirth and Weber's definition of EULER by presenting an EULER compiler and interpreter in Icon. Icon provides all the data structures we need. It provides a garbage collector. And most importantly, we have an Icon system, so we can actually get the compiler and compiled programs to execute.

We generally follow Wirth and Weber's code. We use LL(1) parsing rather than their simple precedence technique, but we preserve the data structures and algorithms of the translation and interpretation code.

First, we will present an informal description of EULER.

Chapter 2 Informal Description of EULER

2.1 Identifiers

Identifiers may be used to name variables, formal parameters, and statement labels. The declaration of formal parameters will be described below in the section on the procedure data type.

2.2 Blocks

A block has the form:

```
begin d; d; d; ... s; s;... s end
```

or

```
begin s; s; ... s end
```

where each d is a declaration and each s is a statement. Blocks permit local definitions of names. As in ALGOL and Pascal, names defined in an enclosing block are known in enclosed blocks. Each name used must have a corresponding declaration. If the name is declared in overlapping scopes, the declaration in the innermost surrounding scope is the corresponding declaration. For example,

```
 1  begin   new x;
 2          label y;
 3          begin   new x;
 4                  new z;
 5                  x;      (* corresponding declaration on line 3 *)
 6                  y;      (* corresponding declaration on line 2 *)
 7                  z       (* corresponding declaration on line 4 *)
 8          end;
 9          y:              (* corresponding declaration on line 2 *)
10                  x       (* corresponding declaration on line 1 *)
11  end
```

The declarations of variables are written:

```
new id
```

The declarations of labels are written:

```
label id
```

Notice that only one identifier is declared per declaration.

A label *L* is *defined*, i.e. bound to a statement *S*, by the form

```
L : S
```

The label must be declared in the beginning of the block in which it is defined.

A variable may be assigned a new value by the expression:

```
id <- expr
```

2.3 Data Types

EULER provides the following data types:

- **number**, integer or real;
- **Boolean**, a logical value;
- **symbol**, a string of characters in quotes;
- **list**, a sequence of elements of any type;
- **reference**, address of a variable or element of a list;
- **label**, a program address;
- **procedure**, a procedure;
- **undefined**, a special value.

2.3.1 number

An integer constant is written as a string of digits, no sign. A real constant is written as two nonempty strings of digits written with a "." in between, optionally followed by an exponent written with an "E" followed by an optional "-" sign followed by an integer.

For example

```
1
1.0
10.0E-1
```

Table 1 numeric functions

isn v	returns true if *v* is a number.
abs e	returns the absolute value of a number.
integer e	rounds the operand of type number to the nearest integer.
- e	returns the negative of *e*.
x + y	addition
x - y	subtraction
x * y	multiplication
x / y	real division
x div y	integer division
x mod y	integer modulus
x ** y	exponentiation
x min y	minimum of the two values
x max y	maximum of the two values
x = y	true if the value of *x* equals the value of *y*
x ~= y	not equal
x < y	less than
x <= y	less than or equal to
x > y	greater than
x >= y	greater than or equal to

2.3.2 Boolean

A Boolean constant is written as "true" or "false".

Table 2 Boolean functions

isb v	returns true if *v* is a Boolean value, false otherwise.
logical v	converts *v* to Boolean (Wirth and Weber don’t define how. We’ll use 0=false, otherwise true)
~ e	returns the logical complement of *e*.
a and b	evaluates a and returns false if *a* is false, otherwise evaluates and returns the value of *b*; notice that the evaluation is short circuited.

Table 2 Boolean functions

a or b	evaluates *a* and returns true if *a* is true, otherwise evaluates and returns the value of *b*; notice that the evaluation is short circuited.
x = y	true if the value of *x* equals the value of *y*
x ~= y	inequality

2.3.3 symbol

A symbol constant is written as a string of characters enclosed in double quotes. An enclosed double quote is written doubled. For example,

```
"""Huh?"" he said."
```

Table 3 symbol functions

isy v	returns true if *v* is a symbol, false otherwise.
x = y	true if the value of *x* equals the value of *y*
x ~= y	inequality

2.3.4 list

There are no list constants.

A list may be constructed by one of the forms:

```
( e1,e2,e3,...,en )
( e1,e2,e3,...,en , )
( )
```

each e_i being an expression. This builds a list of length *n*, the *i*th element is initialized to the corresponding e_i. Notice that you may include a final comma after the last item and that you may create an empty list.

```
list n
```

will create a list of length *n* (*n* is an expression) with each element initialized to the *undefined* value.

The elements of a list are numbered starting at 1. The ith element of a list may be accessed by *e[i]*, where *e* is an expression that evaluates to a list. *e[i]* can also be assigned a value, e.g.

```
a <- (1,2,3);
a[1] <- a[2];
```

will give the list

```
(2,2,3)
```

Table 4 list functions

isli v	return true if *v* is a list, false otherwise.
length e	return the length of the list *e*.
list e	create a list of length *e* with each element initialized to *undef*.
tail e	return the list *e* with the first element removed.
x = y	true if *x* and *y* are pointers to the same list
x ~= y	true if not the same list
e1 & e2	return the list resulting from concatenating the lists *e1* and *e2*

2.3.5 reference

A reference is the address of a variable, a formal parameter, or element of a list. The @ operator will give you a reference. The assignment

```
x <- @ y;
```

will give x a reference to variable y. Thereafter

```
x . <- 5;
```

will assign the value 5 to variable y. The dot is a *dereference* operator. Another example:

```
x <- list 2;
x[1] <- @x[2];
x[1]. <- "garf";
```

will yield a list of the form:

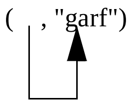


Table 5 reference functions

isr v	returns true if *v* is a reference, false otherwise.

2.3.6 label

An identifier to be used as a label must be declared

```
label id
```

in the declarations part of the block the label occurs in. The label identifier is associated with a statement in the usual way by the form

```
id : statement
```

The label identifier may be used in an expression, creating a label value bound to the statement and the current environment, *e.g.*

```
begin label L;
    ...
    i <- (1,L);
    ...
    goto i[2];
    ...
L:  ...
end;
```

Table 6 label functions

isl v	returns true if *v* contains a label value.

2.3.7 procedure

A procedure is written

```
'expr '
```

or

```
'd; d; ... d; expr '
```

where each d is a formal declaration, written

```
formal id
```

which declares id to be the name of a formal parameter. The quoted procedure yields a procedural value The procedural value must be assigned to a variable to be used.

A procedure call is composed of a variable followed by a list of parameters in parentheses:

```
var ( ... )
```

The variable, *var,* may be a list element.

The occurrence of a procedure causes the created procedural value to be bound within the current environment. For example

```
addx <- ’formal y; x+y’;
```

assigns to variable *addx* a procedure that will take one parameter and return the result of adding the value of variable *x* to it. The instance of variable *x* that will be used is bound at the time the procedure is assigned to *addx*. Even if there is a different variable *x* visible when *addx* is called, it will be the *x* visible where the procedure was created that will be added.

The formal parameters are passed with a kind of call-by-constant-value mechanism. The value of the actual parameter is passed. Within the procedure, the value assigned to a parameter may be used, but it may not be changed.

There is a strange anomaly: if a reference is passed, any access to the formal parameter will access the variable referenced. For example

```
bump <- ’formal x; x <- x + 1’; ...; bump(@a);
```

assigns a procedure to variable bump. When bump is called with a reference as an argument, it increments the value of the referenced variable.

Table 7 procedural functions

isp v	returns true if *v* is a procedure.

2.3.8 undefined

There is a special value representing "undefined". Variables are initialized to it. The constant is written:

```
undef
```

Table 8 functions on the undef type

isu v	returns true if *v* is undefined.

2.4 Control Constructs

Control typically flows from one statement in a block to the next. Other than **procedure calls**, there are two ways to affect the flow of control.

The **if-expression** is written:

if expr **then** expr **else** expr

The **goto expression** is written:

goto expr

The value of the *expr* following the *goto* must be a label.

2.5 Precedence of Operators

The precedences of operators from highest to lowest are

isn isb isl isli isy isp isu length
tail abs integer real logical list
**
* / div mod
+ -
max min
= ~= < <= > >=
~
and
or
&

The precedence levels of expressions may be overridden by grouping subexpressions in rectangular *brackets*. Brackets are in EULER what parentheses are in most languages.

```
a <- b - [ x <- c + d ] * 10;
```

2.6 I/O

out expr	transmits the value of the expression to the output medium.
in	reads a single character, as a symbol, from the input.

2.7 Comments

Comments are enclosed in (* and *) and may be nested.

2.8 Changes from the original EULER

There are a number of differences between the version of EULER presented here and that in the original paper:

- Symbols are an extension of Wirth and Weber's definition. They apparently intended a symbol to be a character rather than a string.
- The equality and inequality (= and ~=) are defined by Wirth and Weber to apply only to integers. We apply them to Booleans and symbols as well.
- The original EULER writes *goto* as *go to*.
- The original EULER used characters not present in ASCII. We have made these substitutions:

ours	original
undef	Ω (omega).

ours	original
and	∧
or	∨
**	↑
div	÷
*	×
~=	≠
<=	≤
>=	≥
~	¬

- The original EULER did not define comments.

2.9 Syntax

Below is a grammar for EULER. It uses approximately the same symbols as the grammar included in the paper, but it is simplified in three ways:

- the simple precedence parser used in the original EULER definition required pairs of names for some nonterminals, e.g. sum and sum-, term and term- as in the following:

sum → sum-	term → term-
sum- → sum- - term	term- → term- * factor
sum- → sum- + term	term- → term- / factor
sum- → - term	term- → term- div factor
sum- → + term	term- → term- mod factor
sum- → term	term- → factor

  Since we are using a more powerful parsing algorithm, we are able to replace *sum-* with *sum*, *term-* with *term* and remove the renaming productions *sum* → *sum-* and *term* → *term-*. We have done so throughout the grammar.

- The original EULER grammar includes productions to define numbers. The semantic actions show how to compute the numeric values of the numbers. In our compiler, the scanner recognizes numbers and the Icon run-time system computes their values. These productions have been removed.

- As mentioned in the discussion of differences, we have made substitutions in order to use the ASCII character set.

Here is the simplified original EULER grammar:

program → block
vardecl → new id

fordecl → formal id
labdecl → label id
var → id
var → var [expr]
var → var .
logval → true
logval → false
reference → @ var
listhead → listhead expr ,
listhead → (
listN → listhead expr)
listN → listhead)
prochead → prochead fordecl ;
prochead → ’
procdef → prochead expr ’
primary → var
primary → var listN
primary → logval
primary → number
primary → symbol
primary → reference
primary → listN
primary → tail primary
primary → procdef
primary → undef
primary → [expr]
primary → in
primary → isb var
primary → isr var
primary → isl var
primary → isli var
primary → isy var
primary → isp var
primary → isu var
primary → abs primary
primary → length var
primary → integer primary
primary → real primary
primary → logical primary
primary → list primary
factor → primary
factor → factor ** primary
term → factor
term → term * factor
term → term / factor
term → term div factor
term → term mod factor
sum → term
sum → + term
sum → - term
sum → sum + term
sum → sum - term
choice → sum
choice → choice min sum
choice → choice max sum
relation → choice
relation → choice = choice
relation → choice ~= choice
relation → choice < choice

relation → choice <= choice
relation → choice > choice
relation → choice >= choice
negation → relation
negation → ~ relation
conjhead → negation and
conj → conjhead conj
conj → negation
disjhead → conj or
disj → disjhead disj
disj → conj
catena → catena & primary
catena → disj
truepart → expr else
ifclause → if expr then
expr → block
expr → ifclause truepart expr
expr → var <- expr
expr → goto primary
expr → out expr
expr → catena
stat → labdef stat
stat → expr
labdef → id :
blokhead → begin
blokhead → blokhead vardecl ;
blokhead → blokhead labdecl ;
blokbody → blokhead
blokbody → blokbody stat ;
block → blokbody stat end

Chapter 3 An EULER Interpreter

3.1 The abstract machine

We will discuss the EULER abstract machine and interpreter before discussing the translator since understanding the translator requires understanding the abstract machine instruction set, but the abstract machine can be understood alone. Nevertheless, in our descriptions of the abstract machine instruction set, we will include short EULER programs and their translations to show how the instructions are used.

3.1.1 *The abstract machine's data structures*

The EULER abstract machine uses the following registers and data structures:

- **S** the stack, containing temporary values during expression evaluation and pointers to activation records containing parameters and variables.
- **i** the stack pointer. In most other systems this would be named *sp*.
- **mp** mark pointer. This points to the position of the top *activation record* in *S*. In many other systems, this would be named *fp*, for frame pointer, since activation records are also called *stack frames*. An older name for stack frame is *mark stack control word*, hence "mark pointer."
- **P** program. This is a list of abstract machine instructions. Each machine instruction is a list. The first element of the list is the *opcode* represented as a string. The following elements, if present, contain the operands.
- **k** program counter, the index of the current instruction in *P*. In most systems this is called *pc*.
- **fct** formal count, a count of the number of formal parameters a procedure requires. It is used to extend a parameter list with undefined values if too few parameters were provided. It has no equivalent in most systems.
- *heap* (it has no name in their interpreter). List structures are allocated dynamically and are freed automatically when no longer accessible. The data structure that allows this is a heap with garbage collection. The heap is hidden since they write their system in EULER and just allocate lists as they need them.

3.1.2 Representation of data types

EULER type	Icon representation	**explanation**
number	Icon’s **integer** or **real**	
Boolean	record Logical(s)	There are exactly two instances of this record. They are assigned to global variables: True:=Logical(“true”) False:=Logical(“false”)
symbol	Icon’s **string**	
list	Icon’s **list**	
reference	record Reference(lst,pos)	*lst* is a list *pos* is an index in the list *lst*
label	record Progref(mix,adr)	*mix* is the index in *S* of the activation record the label was defined in. *adr* is the address of the first instruction of the labeled statement in *P*.
procedure	record procDescr(bln,mix,adr)	*bln* is the block number of the procedure (*i.e.* depth of nesting at which it is to execute). *mix* is the index in *S* of the activation record for the procedure’s surrounding scope. *adr* is the address of the first instruction of the procedure.
undefined	Icon’s **&null**	

3.1.3 Operators

The program

```
begin
out 1+1
end
```

translates into:

```
1 begin
2 number    1
3 number    1
4 +
5 out
6 end
7 halt
```

We will wait until the next example to discuss `begin` and `end`. Here's what the other instructions tell the interpreter to do:

Table 9 Number, out, and halt instructions.

`number` *v*	Push the number *v* onto the stack.
`+`	Add the two top values on the stack. Pop the top value off the stack and add it to the new top of stack. The other binary operations behave similarly to `+`. In all, the top of the stack is the right operand, the value beneath it is the left operand.
`out`	Write the top value on the stack into the output. Don't remove it from the stack. The other unary operations behave similarly to `out`. They replace the top stack element with the result of the operation.
`halt`	Cease execution.

The binary operations are:

```
+ - * / div mod ** min max < <= > >= = ~= &
```

The unary operations are:

```
neg abs integer logical real isn isr isl isli isy
isp isu ~ length tail list value
```

Notice that most of the binary and unary abstract machine operations have exactly the same names as the corresponding operations in EULER. We use this fact to simplify the translator. There are two exceptions in the list: Unary minus is translated into a `neg` instruction, "`-`" having already been used for the binary minus. The operation "`value`" is usually implicit in the context in the source program and not usually made explicit with the suffix "." operator. EULER's unary plus operator has no abstract machine operation because it performs no operation.

The instructions that load values are:

Table 10 Load instructions.

`number` *v*	Push the number v

Table 10 Load instructions.

`logval` *v*	Push the logical value v
`undef`	Push the undefined value
`symbol` *v*	Push the symbol (string) v
`in`	Push the next symbol (character) read from the input

3.1.4 Blocks, variables, and assignments

The program

```
begin new x; new y;
x <- 1;
y <- x+1;
out y
end
```

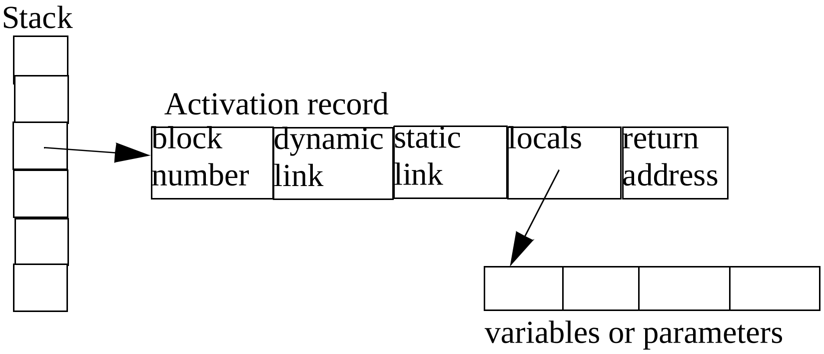
translates into

```
 1 begin
 2 new
 3 new
 4 @        1,1
 5 number   1
 6 <-
 7 ;
 8 @        2,1
 9 @        1,1
10 value
11 number   1
12 +
13 <-
14 ;
15 @        2,1
16 value
17 out
18 end
19 halt
```

Table 11 Block, variable, and assignment instructions.

`begin`	Push a new activation record onto the stack. Assign *mp* the position of the activation record. In real implementations, the fields of the activation record would be on the stack itself. Here the activation record is a list and the stack *S* holds a pointer to it.
`end`	Pop an activation record off the stack. The top of the stack has the value computed by the block. The next element of the stack has a pointer to the block's activation record. The value returned by the block is pushed down, replacing the pointer to the activation record.
`new`	A `new` instruction is generated for each variable declared. An activation record contains a list with one element for each variable. When the activation record is created, the list is empty. The `new` instruction creates a variable by putting another element on the variable list and initializing it to undefined.
`@  on,bn`	The `@` instruction creates a reference to a variable and pushes it on the stack. The variable is at position *on* (ordinal number) in the list of variables in the surrounding block with number *bn*. A reference is an internal data type that allows the value of a variable to be fetched and a new value to be assigned. (We will discuss references below.)
`value`	The `value` instruction examines the top element of the stack. If it is a reference, then it takes the reference off the top of the stack and replaces it with the value of the referenced variable (*i.e. dereferencing* it) and examines it again. If it is a procedural value, it calls the procedure passing it an empty parameter list, *deproceduring* it. The `value` instruction will first try to *dereference* and then try to *deprocedure*, in that order, accomplishing neither, either, or both. Any value other than a reference or a procedural value is left alone.
`<-`	The assignment instruction, `<-`, finds a value on top of the stack and a reference immediately beneath it. The assignment instruction pops value and the reference off the stack, assigns the value to the variable referenced, and pushes the value back on the stack.
`;`	The `;` instruction pops the top value off the stack. EULER is an expression language where every statement produces a value. When executing a statement sequence, the value of each statement but the last must be popped off the stack.

The activation records for blocks are as follows:



where

block number	is the depth of nesting of the block and is used to find the correct activation record for a variable or parameter.
dynamic link	a copy of *mp* at the time this block was entered.
static link	points to the position in the *S* of the activation record of the surrounding block. Searches for non-local variables use the static links.
locals	is a list of storage for procedure (*formal*) parameters and local (*new*) variables.
return address	has the index in the *P* array to return to, for procedure activation records. For begin/end block activation records, this field is omitted.

Activation records are chained together. The dynamic chain links each activation record to its caller, indicated by *mp* at the time the activation record was created. The static chain links an activation record to the activation record for the surrounding scope. Each activation record, therefore, contains two link fields, one for each chain. The activation record pushed by `begin` has both its static and dynamic link initialized to the same value, the value of *mp* on entry.

A variable always pushes a reference onto the stack. The context in which the variable is used can cause its value to be fetched. In fact, almost everywhere the compiler generates a `value` instruction following the variable. The two exceptions are on the left of an assignment operator, `<-`, and as the operand of the `@`, both of which suppress the generation of the `value` instruction.

A reference is an internal data type in the run-time system. Internally, a reference contains a pointer to a list, *L*, and an index, *j*. The `value` instruction fetches the contents *L[j]*. The <- instruction changes it. The “`@ on,bn`” instruction searches the static chain for the activation record with block number *bn* and pushes a reference to element *on* of its variable list.

3.1.5 Conditionals

The program

```
begin new x; new y; new z;
  if x and y or ~z then 1 else 2
end
```

translates into

```
 1 begin
 2 new
 3 new
 4 new
 5 @       1,1
 6 value
 7 and     10
 8 @       2,1
 9 value
10 or      14
11 @       3,1
12 value
13 ~
14 then    17
15 number  1
16 else    18
17 number  2
18 end
19 halt
```

(If you try to run this, it will terminate with an undefined variable error. You might want to try it out with assignments of *true*s and *false*s to the variables.)

The significant new instructions here are jump instructions.

Table 12 Jump instructions

else d	The `else` instruction jumps unconditionally to instruction *P[d] i.e.* it sets the program counter *k* to *d*.. It probably should have been named `jump`.
then d	The `then` instruction pops the top value from the stack. If the value popped was *false*, it jumps to instruction *P[d]*, *i.e.* it sets the program counter *k* to *d*. If the value popped was *true*, it falls through to the next instruction.

Table 12 Jump instructions

and d	The `and` instruction is a conditional branch designed to short-circuit conditional expressions. The `and` instruction is generated after the left operand of an *and* operator and before the right. It tests the top value on the stack, the value of the left operand. If the value is *false*, it is clear that the value of the entire expression will be *false*; the `and` instruction sets the program counter *k* to *d*, jumping to the instruction that follows the right subexpression with the false still on the top of the stack. If the value atop the stack is *true*, then the value of the expression will be the value of the right hand side. The `and` instruction pops the top element off the stack and falls through to evaluate the right hand side.
or d	The `or` instruction is like the `and` instruction except that it reverses the significance of *true* and *false*. If the top of the stack is *true*, the instruction sets the program counter *k* to *d*. If the top value is *false*, it pops the value and falls through.

3.1.6 Labels and gotos

The program

```
begin label L1; label L2;
   goto L2;
L1:
   goto L1;
L2: 0
end
```

translates into

```
 1 begin
 2 label     10,1
 3 value
 4 goto
 5 ;
 6 label     6,1
 7 value
 8 goto
 9 ;
10 number    0
11 end
12 halt
```

Table 13 Label and goto instructions.

`label` *`pa,bn`*	The `label` instruction pushes a program address value onto the stack. In a block structured language, a label must contain both an instruction address and an environment. Operand *pa* gives the index in *P* of the instruction. Operand *bn* gives the number of the block in which the label is defined. In the program address value, *bn* is translated into the index in the stack of the activation record with that block number.
`goto`	The `goto` instruction pops a program address value off the top of the stack, assigns its *pa* value to *k* (the program counter) and sets *mp* and *i* (the stack pointer) from its environment performing a block-structured goto.

The `value` instruction following the label instruction performs no operation and could be eliminated. It is there as a consequence of how the translator handles variable identifiers.

3.1.7 Procedures calls and lists

The program

```
begin new bump; new a;
 bump <- ’formal x; x <- x + 1’;
 a <- 1;
 bump(@a);
 out a
end
```

translates into

```
 1 begin
 2 new
 3 new
 4 @        1,1
 5 proc     16
 6 formal
 7 @        1,2
 8 value
 9 @        1,2
10 value
11 value
12 number   1
13 +
14 <-
15 endproc
```

```
16 <-
17 ;
18 @        2,1
19 number   1
20 <-
21 ;
22 @        1,1
23 @        2,1
24 )        1
25 call
26 ;
27 @        2,1
28 value
29 out
30 end
31 halt
```

Table 14 Procedures, calls, and lists.

`proc` *pa*	The `proc` instruction pushes a procedural value on the stack and then jumps to the instruction at location *pa*. The instructions for the procedure immediately follow the `proc` instruction, so the jump is necessary to get past them. The procedural value must contain both the address of the procedure's code and also an environment, the value to be placed in the static link field of the procedure's activation record. Since the procedure is local to the current environment, the value of *mp* is saved as the environment. The procedural value also holds the block number, that is, depth of nesting, of the procedure.
`)` *n*	The `)` instruction creates an initialized list. It creates a list of *n* elements and fills it with *n* elements removed from the stack. The top element of the stack becomes the rightmost, *n*th, element of the list.

Table 14 Procedures, calls, and lists.

`call`	The `call` instruction calls a procedure. The top element on the stack is a list, the actual parameter list of the procedure. The next-to-top element is a procedural value. The `call` instruction pops both elements, pushes an activation record for the procedure, sets *fct* to zero, and jumps to the procedure's entry. The fields of the activation record are set as follows: 1 The *block number* is set from the procedural value. 2 The *dynamic link* is set to mp. 3 The *static link* is set from the environment field of the procedural value. 4 The *locals* list is set to the actual parameter list. 5 The *return address* is set to the program counter, *k*.
`formal`	One `formal` instruction is generated for each formal parameter declared. The purpose of the `formal` instruction is to extend the actual parameter list if it is shorter than the number of formal parameters. The instruction increments *fct*. If *fct* is larger than the length of the parameter list—and it can only be larger by one—an undefined value is appended to the end of the actual parameter list to make them equal.
`endproc`	The `endproc` instruction behaves much like the `end` instruction. It finds the value computed by the procedure body on top of the stack and the procedure's activation record just beneath it. It exits the procedure much like `end` exits a block, popping the value and replacing the activation record with it. The *mp* register is set to point back to the caller's activation record. The program counter, *k*, is set to the return address.

3.1.8 Subscripting

The program

```
begin new x;
x <- (1,2);
out x[1]
end
```

translates into

```
1 begin
2 new
3 @        1,1
4 number   1
5 number   2
```

```
 6 )       2
 7 <-
 8 ;
 9 @       1,1
10 number  1
11 ]
12 value
13 out
14 end
15 halt
```

Table 15 Subscripting instruction.

`]`	The `]` instruction performs subscripting. It should find a numeric value, *j*, on top of the stack and a reference to a variable containing a list, *L*, just beneath it on the stack. It removes both and pushes back a reference to the indicated element of the list, *L[j]*.

3.2 The interpreter

The interpreter consists of the usual instruction fetch/execute cycle implemented as a case expression in a loop. The Icon code for the EULER interpreter follows.

Notes on the interpreter:

Line 2 declares the abstract machine's registers and storage. S is the stack. P, the program array, is declared in the translator.

Lines 4-6 declares EULER's new data types.

Line 4 declares the reference data type. Field *lst* is a list; field *pos* is an integer giving a position in the list.

Line 5 declares a program reference, which is to say, a label. Field *mix* is the index in *S* of the activation record the label is bound within. Field *adr* is the address in *P* of the instruction. When the program goes to the program reference, *mix* is loaded into both the frame pointer, *mp*, and the stack pointer, *i*; *adr* is loaded into the program counter, *k*.

Line 6 declares a procedure descriptor, more commonly called a *closure*. Field *bln* is the block number of the procedure; *mix*, the index in *S* of its activation record; *adr*, the address of its first instruction in *P*.

Lines 8-17 construct a reference from a block number and an ordinal number.

Lines 19-28 construct a program reference from a program address (an index in *P*) and a block number.

Lines 30-33 *dereference*—fetch the value to which a reference points. If the operand is not a reference, it is returned unaltered.

Lines 35-42 assign a value to a referenced variable.

Lines 44-391 are the interpreter itself.

Line 46 allocates a fixed-sized stack. This follows Wirth and Weber's code. It might be better to try using Icon's *put* and *pull*.

Line 47 starts the stack pointer at the bottom of the stack.

Lines 48-49 push an initial activation record (block number zero) on the stack. This is not done in the original EULER paper, but `begin` needs it.

Line 50 sets the program counter to start execution at the first instruction.

Lines 51-394 is the main instruction fetch/execute loop. *P[k]* is the instruction. *P[k][1]* is the op-code. Notice that the program counter (*k*) is incremented at the end of the instruction execution at line 393. Jumps will bypass the increment by executing *next*.

Lines 54-109 are numeric binary operators.

Lines 110-134 are unary operators.

Lines 151-182 are type test operators.

Lines 183-198 are primitive versions of the I/O operators. They need improvement.

Lines 199-230 are numeric relational operators.

Lines 231-238 test equality or inequality. In the original EULER, these are numeric comparisons, but we've extended them to perform an identity test so that they can check whether two lists are actually the same list or whether two symbols are the same strings. They really should be extended further to test references, program references, and procedural values for equality.

Lines 239-250 are conditional jumps.

Lines 290-293 load constant values on the stack. Programmed in Icon, only one such instruction is really needed.

Lines 306-308 allocate a new local variable and initialize it to *undef*.

Lines 309-312 declare a new formal parameter. Variable *fct* keeps a count of the number of formal parameters the procedure has declared. If *fct* is greater than the length of the formal parameter list, a formal parameter is allocated and initialized to *undef*.

```
 1 #      Euler Interpreter
 2 global S,k,i,mp,fct
 3
 4 record Reference(lst,pos)
 5 record Progref(mix,adr)
 6 record procDescr(bln,mix,adr)
 7
 8 procedure reference(on,bn)
 9 local j
10 j := mp
11 while j>0 do {
12        if S[j][1] = bn then return Reference(S[j][4],on)
13        j := S[j][3]#static link
14 }
15 RTError("dangling reference")
16 fail
17 end
18
19 procedure progref(pa,bn)
20 local j
21 j := mp
22 while j>0 do {
23        if S[j][1] = bn then return Progref(j,pa)
24        j := S[j][3]#static link
25 }
26 RTError("dangling reference")
27 fail
28 end
```

```

procedure deref(x)
if type(x) ~== "Reference" then return x
return x.lst[x.pos]
end

procedure assignThroughRef(x,v)
local j
if type(x) ~== "Reference" then {
    RTError("reference needed on left of ’<-’")
    fail
}
return x.lst[x.pos] := v
end

procedure interpreter()
local l,r,t
S := list(500)
i := 1
S[1] := [0,0,0,[]]#outer, empty activation record
mp := 1
k := 1
repeat {
  if k>*P then return
  case P[k][1] of {
"+": {
    if not (l:=numeric(S[i-1])) then
            return RTError("numeric required")
    if not (r:=numeric(S[i])) then
            return RTError("numeric required")
    i -:= 1
    S[i] := l + r
    }
"-":  {
    if not (l:=numeric(S[i-1])) then
            return RTError("numeric required")
    if not (r:=numeric(S[i])) then
            return RTError("numeric required")
    i -:= 1
    S[i] := l - r
    }
"*": {
    if not (l:=numeric(S[i-1])) then
            return RTError("numeric required")
    if not (r:=numeric(S[i])) then
            return RTError("numeric required")
    i -:= 1
    S[i] := l * r
    }
"/":  {
    if not (l:=real(S[i-1])) then
            return RTError("numeric required")
    if not (r:=real(S[i])) then
            return RTError("numeric required")
    i -:= 1
    S[i] := l / r
    }
"div":{
    if not (l:=integer(S[i-1])) then
```

```
            return RTError("numeric required")
    if not (r:=integer(S[i])) then
            return RTError("numeric required")
    i -:= 1
    S[i] := l / r
    }
"mod":{
    if not (l:=integer(S[i-1])) then
            return RTError("numeric required")
    if not (r:=integer(S[i])) then
            return RTError("numeric required")
    i -:= 1
    S[i] := l % r
    }
"**":{
    if not (l:=numeric(S[i-1])) then
            return RTError("numeric required")
    if not (r:=numeric(S[i])) then
            return RTError("numeric required")
    i -:= 1
    S[i] := l ^ r
    }
"neg":{
    if not (r:=numeric(S[i])) then
            return RTError("numeric required")
    S[i] := - r
    }
"abs":{
    if not (r:=numeric(S[i])) then
            return RTError("numeric required")
    S[i] := abs(r)
    }
"integer":{
    if not (r:=numeric(S[i])) then
            return RTError("numeric required")
    S[i] := integer(r)
    }
"logical":{
    if not (r:=numeric(S[i])) then
            return RTError("numeric required")
    S[i] := if r ~= 0 then True else False
    }
"real":{
    if type(r:=S[i])~=="Logical" then
            return RTError("logical required")
    S[i] := if r === True then 1 else 0
    }
"min":{
    if not (l:=numeric(S[i-1])) then
            return RTError("numeric required")
    if not (r:=numeric(S[i])) then
            return RTError("numeric required")
    i -:= 1
    S[i] := if l < r then l else r
    }
"max":{
    if not (l:=numeric(S[i-1])) then
            return RTError("numeric required")
    if not (r:=numeric(S[i])) then
```

```
            return RTError("numeric required")
    i -:= 1
    S[i] := if l > r then l else r
    }
"isn":{
    r:=deref(S[i])
    S[i] := if numeric(r) then True else False
    }
"isb":{
    r:=deref(S[i])
    S[i] := if type(r)=="Logical" then True else False
    }
"isr":{
    r:=deref(S[i])
    S[i] := if type(r)=="Reference" then True else False
    }
"isl":{
    r:=deref(S[i])
    S[i] := if type(r)=="Progref" then True else False
    }
"isli":{
    r:=deref(S[i])
    S[i] := if type(r)=="list" then True else False
    }
"isy":{
    r:=deref(S[i])
    S[i] := if type(r)=="string" then True else False
    }
"isp":{
    r:=deref(S[i])
    S[i] := if type(r)=="procDescr" then True else False
    }
"isu":{
    r:=deref(S[i])
    S[i] := if /r then True else False
    }
"in":{
    i+:=1
    S[i]:=reads()
    }
"out":{
    r:=deref(S[i])
    case type(r) of {
    "Logical": write(r.s)
    "null": write("undef")
    "Reference":write("Reference(",image(r.lst),",",r.pos,")")
    "Progref":write("Program_Reference(",r.mix,",",r.adr,")")
    "procDescr":write("Procedure_Descriptor(",
                    r.bln,",",r.mix,",",r.adr,")")
    default: write(r)
    }
    }
"<=":{
    if not (l:=numeric(S[i-1])) then
            return RTError("numeric required")
    if not (r:=numeric(S[i])) then
            return RTError("numeric required")
    i -:= 1
    S[i] := if l <= r then True else False
```

```
	}
"<":{
	if not (l:=numeric(S[i-1])) then
		return RTError("numeric required")
	if not (r:=numeric(S[i])) then
		return RTError("numeric required")
	i -:= 1
	S[i] := if l < r then True else False
	}
">=":{
	if not (l:=numeric(S[i-1])) then
		return RTError("numeric required")
	if not (r:=numeric(S[i])) then
		return RTError("numeric required")
	i -:= 1
	S[i] := if l >= r then True else False
	}
">":{
	if not (l:=numeric(S[i-1])) then
		return RTError("numeric required")
	if not (r:=numeric(S[i])) then
		return RTError("numeric required")
	i -:= 1
	S[i] := if l > r then True else False
	}
"=":{
	i -:= 1
	S[i] := if S[i] === S[i+1] then True else False
	}
"~=":{
	i -:= 1
	S[i] := if S[i] ~=== S[i+1] then True else False
	}
"and":{
	if type(r:=S[i])~=="Logical" then
		return RTError("logical required")
	if r===True then i-:=1
	else { k:=P[k][2]; next }
	}
"or":{
	if type(r:=S[i])~=="Logical" then
		return RTError("logical required")
	if r===True then { k:=P[k][2]; next }
	else i-:=1
	}
"~":{
	if type(r:=S[i])~=="Logical" then
		return RTError("logical required")
	S[i] := if r===True then False else True
	}
"then":{
	if type(r:=S[i])~=="Logical" then
		return RTError("logical required")
	i-:=1
	if r===False then { k:=P[k][2]; next }
	}
"else":{
	k:=P[k][2]
	next
```

```
    }
"length": {
    r:=deref(S[i])
    if type(r)~=="list" then
            return RTError("list required")
    S[i] := *r
    }
"tail": {
    if type(r:=S[i])~=="list" then
            return RTError("list required")
    if *r<1 then
            return RTError("non-empty list required")
    S[i] := r[2:0]
    }
"&":{
    if not (type(l:=S[i-1])==type(r:=S[i])=="list") then
            return RTError("list required")
    i -:= 1
    S[i] := l ||| r
    }
"list":{
    if not (r:=numeric(S[i])) then
            return RTError("numeric required")
    S[i] := list(r)
    }
"number"|"logval"|"symbol" : {
    i +:= 1
    S[i] := P[k][2]
    }
"undef": {
    i +:= 1
    S[i] := &null
    }
"label": {
    i +:= 1
    S[i] := progref(P[k][2],P[k][3])
    }
"@":{
    i +:= 1
    S[i] := reference(P[k][2],P[k][3])
    }
"new":{
    put(S[mp][4],&null)
    }
"formal": {
    fct +:= 1
    if fct > *S[mp][4] then put(S[mp][4],&null)
    }
"<-":{
    i -:= 1
    S[i] := assignThroughRef(S[i],S[i+1]) | fail
    }
";": {
    i -:= 1
    }
"]": {
    if not (r:=numeric(S[i])) then
            return RTError("numeric required")
    if r <= 0 then
```

```
                return RTError("subscript must be positive")
        i -:= 1
        l := deref(S[i])
        if type(l)~=="list" then
                return RTError("list required")
        if r > *l then return RTError("subscript too large")
        S[i] := Reference(l,r)
        }
"begin": {
        i +:= 1
        S[i] := [S[mp][1]+1,mp,mp,[]]
        mp := i
        }
"end":{
        t := S[mp][2]
        S[mp] := S[i]
        i := mp
        mp := t
        }
"proc":{
        i +:= 1
        S[i] := procDescr(S[mp][1]+1,mp,k)
        k := P[k][2]
        next
        }
"value": {
        S[i] := t := deref(S[i])
        if type(t)=="procDescr" then {
                fct := 0
                S[i] := [t.bln,mp,t.mix,[],k]
                mp := i
                k := t.adr
                }
        }
"call": {
        i -:= 1
        t := deref(S[i])
        if type(t)~=="procDescr" then
                return RTError("procedure required")
        fct := 0
        S[i] := [t.bln,mp,t.mix,S[i+1],k]
        mp := i
        k := t.adr
        }
"endproc": {
        k := S[mp][5]
        t := S[mp][2]
        S[mp] := S[i]
        i := mp
        mp := t
        }
"halt":{
        break
        }
"goto":{
        if type(S[i])~=="Progref" then
                return RTError("label required")
        mp := S[i].mix
        k := S[i].adr
```

```
        i := mp
        next
        }
")": {
        i +:= 1
        r := S[i-P[k][2]:i]
        i -:= P[k][2]
        S[i] := r
        }
  }
        k+:=1
}
return
end

procedure RTError(s)
stop(k," ",P[k][1]," --- ",s)
end

```

Chapter 4 The EULER Translator

4.1 Parser

Here is a grammar for EULER. The many levels of operators in EULER and the labeled statements caused the major difficulties in putting the grammar into LL(1) form.

```
start : program .
program = block ENDPROG!.
vardecl = new id NEWDECL! .
fordecl = formal id FORMALDECL! .
labdecl = label id LABELDECL! .
var = id VARID! {  "[" expr "]" SUBSCR! |  "." DOT! } .
logval = true LOGVALTRUE! .
logval = false LOGVALFALSE! .
number = realN | integerN.
reference = "@" var REFERENCE! .
#     listhead -> "(" LISTHD1!
#     listhead -> listhead expr "," LISTHD2!
#     listN -> listhead ")" LISTN1!
#     listN -> listhead expr ")" LISTN2!
listN = "(" LISTHD1! ( ")" LISTN1! | expr listTl ) .
listTl = ")" LISTN2! | "," LISTHD2! ( expr listTl | ")" LISTN1! ) .
prochead = "”" PROCHD! {  fordecl ";" PROCFORDECL! } .
procdef = prochead expr "”" PROCDEF! .
primary = var ( listN CALL! | VALUE!) | primary1 .
primary1 = logval LOADLOGVAL! | number LOADNUM! |
      symbol LOADSYMB!| reference |
      listN | tail primary UOP! | procdef |
      undef LOADUNDEF! | "[" expr "]" PARENS! | in INPUT! |
      isb var UOP! | isn var UOP! | isr var UOP! |
      isl var UOP! | isli var UOP! | isy var UOP! |
      isp var UOP! | isu var UOP! | abs primary UOP! |
      length var UOP! | integer primary UOP! |
      real primary UOP! | logical primary UOP! | list primary UOP! .
factor = primary factortail.
factortail = { "**" primary BOP! } .
```

```
term = factor termtail.
termtail = { "*" factor BOP! | "/" factor BOP! |
            div factor BOP! | mod factor BOP! } .
sum = ("+" term UPLUS! | "-" term NEG! | term) sumtail.
sumtail = { "+" term BOP! | "-" term BOP! } .
choice = sum choicetail.
choicetail = { min sum BOP! | max sum BOP! } .

relation = choice relationtail.
relationtail = [ "=" choice BOP! | "~=" choice BOP!
    | "<" choice BOP! | "<=" choice BOP!
    | ">" choice BOP! | ">=" choice BOP! ] .

negation =  "~" relation UOP! | relation .
conj = negation conjtail.
conjtail = [ and CONJHD! conj CONJ! ].
disj = conj disjtail.
disjtail = [ or DISJHD! disj DISJ! ] .
catenatail = { "&" primary BOP! }.

truepart = expr else TRUEPT! .
ifclause = if expr then IFCLSE! .

expr = var exprtail | expr1.
exprtail = "<-" expr BOP! |
    ( listN CALL! | VALUE!)
    factortail
    termtail
    sumtail
    choicetail
    relationtail
    conjtail
    disjtail
    catenatail .

expr1 = block .
expr1 = ifclause truepart expr IFEXPR! .
expr1 = goto primary UOP! .
expr1 = out expr UOP! .
expr1 =primary1
    factortail
    termtail
    sumtail
```

```
        choicetail
        relationtail
        conjtail
        disjtail
        catenatail .
    expr1 =  ( "+" term UPLUS! | "-" term NEG! )
        sumtail
        choicetail
        relationtail
        conjtail
        disjtail
        catenatail .
    expr1 = "~" relation UOP! conjtail disjtail catenatail .

    stat = expr1
        | id ( ":" LABDEF! stat LABSTMT!
                | VARID! {  "[" expr "]" SUBSCR! |  "." DOT! }
                        exprtail ) .

    block = begin BEGIN!
                { vardecl ";" BLKHD! | labdecl ";" BLKHD!}
                stat { ";" BLKBODY! stat } end BLK! .
```

4.2 Translator

The translator uses a semantics stack. Whenever the parser recognizes a token, it pushes it onto the semantics stack. Whenever the parser encounters an action symbol, it executes a routine to generate code. The action routine removes a fixed number of values from the semantics stack, performs its action, and pushes a single value back on the semantics stack. The general format of an action routine is:

```
1 procedure <<Action name>>()
2 V:=popSem(<<Length of right hand side>>)
3 if errorFound:=anyError(V) then return pushSem(errorFound)
4 <<Body of action>>
5 pushSem(<<Semantic value of left hand side>>)
6 return
7 end
```

Line 2 removes values from the semantics stack, placing them in the V list. Line 3 tests to see if any subphrase was in error, and if so skips generating code and propagates the error upwards. Line 4 represents all the lines of the body of the action routine. Line 5 pushes the semantics value computed for the entire phrase back on the stack.

The following lists the meanings of some of the variables used in the code:

V	array of semantic values of symbols on RHS, e.g.

relation	→	choice	<=	choice
		V[1]	V[2]	V[3]

P	program produced by translator
k	index into P
N	list of identifiers & associated data
n	index into N
m	index into N
bn	block number
on	ordinal number

The translator places the code it generates in list *P*. The code is generated strictly left-to-right, bottom-up. Each generated instruction is itself a list. The first element of the instruction is the name of the instruction—represented as a character string. Any subsequent elements are the operand fields.

There are various forms of jump instructions that jump forward in the code. Their destination is not known when the instruction is generated, so the destination is *back patched* into the instructions later. In some cases, like `or`, `and`, `then`, and `else`, the destination field is initialized to *&null*, the address of the instruction is pushed on the semantics stack as the value of the phrase that generated it, and the actual address is inserted by the action routine for the enclosing phrase. Instruction `or` is generated in action routine DISJHD (lines 303-309) and backpatched in DISJ (lines 311-317). Instruction `and` is handled similarly to *or*. Instruction `then` is generated in action routine IFCLSE (lines 327-333) and `else` in TRUEPT (lines 319-325); they are both backpatched in IFEXPR (lines 335-342).

In the case of *var* -> *id* where the identifier names a label, the translator generates a `label` instruction. The `label` instruction must contain the program address of the label, but the label might not be defined yet. In that case, the `label` instruction is placed on a linked list attached to the symbol table entry for the label. When the label is defined, all instructions on the list are patched to point to its location. The label is entered into the symbol table in action routine LABELDECL (lines 66-73). The label instruction is generated in VARID (lines 75-99). The label is defined in LABDEF (lines 351-373), where the address of the label is found and any forward references to it are backpatched.

The translator keeps its symbol table in list *N*. The symbol table is searched by a linear scan.

Each symbol entry has four fields:

id	bn	on	type

where

id	is the name of the entry, a string.
bn	is the block number where the entry is defined.
on	is an ordinal number—for a variable or a formal parameter, its position in the list of formals and variables in its block; for a label, either its position in the P array, or the position of the first instruction in a list of forward references to the label.
type	is “formal” for a formal parameter; “new” for a variable; “label” for a label that has already been assigned a position in the array P; and *undef* for a label that is not yet defined.

To see symbols being inserted into N, see action routines NEWDECL, FORMALDECL, and LABELDECL (lines 44-73). To see symbols being consulted in the symbol table, see action routines VARID (lines 75-99) and LABDEF (lines 351-373).

The symbol table is block-structured. At any point in the program, each enclosing block has a contiguous section of the N stack containing its symbols. Each section begins with a marker

undef	link

where link points to (actually, is the index of) the marker for the surrounding block. Variable *m* is the index of the top marker on the N stack.

To see markers being inserted in N, look at the action routines PROCHD (lines 175-185) and BEGIN (lines 375-386). To see markers being removed, look at PROCDEF (lines 187-197) and BLK (lines 403-412).

The following table shows the original grammar with the associated action routines and where they occur in the code.

Table 16 Action routines

Production	Action	Lines
program → block	ENDPROG	39-42
vardecl → new id	NEWDECL	44-53
fordecl → formal id	FORMALDECL	55-64
labdecl → label id	LABELDECL	66-73
var → id	VARID	75-99
var → var [expr]	SUBSCR	101-107
var → var .	DOT	109-115
logval → true	LOGVALTRUE	117-122

Table 16 Action routines

Production	Action	Lines
logval → false	LOGVALFALSE	124-129
reference → @ var	REFERENCE	131-136
listhead → listhead expr ,	LISTHD2	138-143
listhead → (	LISTHD1	145-150
listN → listhead expr)	LISTN2	152-158
listN → listhead)	LISTN1	160-166
prochead → prochead fordecl ;	PROCFORDECL	168-173
prochead → ’	PROCHD	175-185
procdef → prochead expr ’	PROCDEF	187-197
primary → var	VALUE	199-205
primary → var listN	CALL	207-213
primary → logval	LOADLOGVAL	215-221
primary → number	LOADNUM	223-229
primary → symbol	LOADSYMB	231-237
primary → reference		
primary → listN		
primary → tail primary	UOP	256-262
primary → procdef		
primary → undef	LOADUNDEF	239-242
primary → [expr]	PARENS	244-249
primary → in	INPUT	251-254
primary → isb var	UOP	256-262
primary → isr var	UOP	256-262
primary → isl var	UOP	256-262
primary → isli var	UOP	256-262
primary → isy var	UOP	256-262
primary → isp var	UOP	256-262
primary → isu var	UOP	256-262
primary → abs primary	UOP	256-262

Table 16 Action routines

Production	Action	Lines
primary → length var	UOP	256-262
primary → integer primary	UOP	256-262
primary → real primary	UOP	256-262
primary → logical primary	UOP	256-262
primary → list primary	UOP	256-262
factor → primary		
factor → factor ** primary	BOP	264-270
term → factor		
term → term * factor	BOP	264-270
term → term / factor	BOP	264-270
term → term div factor	BOP	264-270
term → term mod factor	BOP	264-270
sum → term		
sum → + term	UPLUS	272-277
sum → - term	NEG	279-285
sum → sum + term	BOP	264-270
sum → sum - term	BOP	264-270
choice → sum		
choice → choice min sum	BOP	264-270
choice → choice max sum	BOP	264-270
relation → choice		
relation (choice = choice	BOP	264-270
relation → choice ~= choice	BOP	264-270
relation → choice < choice	BOP	264-270
relation → choice <= choice	BOP	264-270
relation → choice > choice	BOP	264-270
relation → choice >= choice	BOP	264-270
negation → relation		

Table 16 Action routines

Production	Action	Lines
negation → ~ relation	UOP	256-262
conjhead → negation and	CONJHD	287-293
conj → conjhead conj	CONJ	295-301
conj → negation		
disjhead → conj or	DISJHD	303-309
disj → disjhead disj	DISJ	311-317
disj → conj		
catena → catena & primary	BOP	264-270
catena → disj		
truepart → expr else	TRUEPT	319-325
ifclause → if expr then	IFCLSE	327-333
expr → block		
expr → ifclause truepart expr	IFEXPR	335-342
expr → var <- expr	BOP	264-270
expr → goto primary	UOP	256-262
expr → out expr	UOP	256-262
expr → catena		
stat → labdef stat	LABSTMT	344-349
stat → expr		
labdef → id :	LABDEF	351-373
blokhead → begin	BEGIN	375-386
blokhead → blokhead vardecl ;	BLKHD	388-393
blokhead → blokhead labdecl ;	BLKHD	388-393
blokbody → blokhead		
blokbody → blokbody stat ;	BLKBODY	395-401
block → blokbody stat end	BLK	403-412

The following is the Icon code for the EULER translator:

```
1 #EULER semantics routines
2
```

```
record Logical(s)
global True, False
global P,N,n,m,bn,on,V,semantics

procedure initTrans()
P:=[]
N:=list(100)
bn:=0
on:=0
n:=0
m:=0
True := Logical("true")
False := Logical("false")
return
end

procedure pushCTError(M[])
every writes(!M)
write()
push(semanticsStack,&null)
return
end

procedure showCode()
local i,h
h:=*string(*P)
every i:=1 to *P do {
 writes(right(i,h), " ", left(P[i][1],10))
 every writes(image(P[i][2 to *P[i]-1]),",")
 if P[i][1]=="logval" then writes(P[i][2].s)
 else writes(image(P[i][1<*P[i]]))
 write()
}
return
end

procedure ENDPROG()
put(P,["halt"])
return
end

procedure NEWDECL()
V:=popSem(2)
if errorFound:=anyError(V) then return pushSem(errorFound)
put(P,["new"])
on+:=1
n+:=1
N[n] := [V[2].body,bn,on,"new"]
pushSem(&null)
return
end

procedure FORMALDECL()
V:=popSem(2)
if errorFound:=anyError(V) then return pushSem(errorFound)
put(P,["formal"])
on+:=1
n+:=1
N[n] := [V[2].body,bn,on,"formal"]
```

```
pushSem(&null)
return
end

procedure LABELDECL()
V:=popSem(2)
if errorFound:=anyError(V) then return pushSem(errorFound)
n+:=1
N[n] := [V[2].body,bn,&null,&null]
pushSem(&null)
return
end

procedure VARID()
local t
V:=popSem(1)
if errorFound:=anyError(V) then return pushSem(errorFound)
t:=n
while t>=1 do {
 if N[t][1]===V[1].body then break
 t -:= 1
}
if t<1 then
 return pushCTError("identifier ",V[1].body," undeclared")
if N[t][4]==="new" then {
 put(P, ["@",N[t][3],N[t][2]] )
} else if N[t][4]==="label" then {
 put(P, ["label",N[t][3],N[t][2]] )
} else if N[t][4]==="formal" then {
 put(P, ["@",N[t][3],N[t][2]] )
 put(P, ["value"])
} else {
 put(P, ["label",N[t][3],N[t][2]] )
 N[t][3] := *P
}
pushSem(&null)
return
end

procedure SUBSCR()
V:=popSem(4)
if errorFound:=anyError(V) then return pushSem(errorFound)
put(P, ["]"] )
pushSem(&null)
return
end

procedure DOT()
V:=popSem(2)
if errorFound:=anyError(V) then return pushSem(errorFound)
put(P, ["value"] )
pushSem(&null)
return
end

procedure LOGVALTRUE()
V:=popSem(1)
if errorFound:=anyError(V) then return pushSem(errorFound)
pushSem(True)
```

```
return
end

procedure LOGVALFALSE()
V:=popSem(1)
if errorFound:=anyError(V) then return pushSem(errorFound)
pushSem(False)
return
end

procedure REFERENCE()
V:=popSem(2)
if errorFound:=anyError(V) then return pushSem(errorFound)
pushSem(&null)
return
end

procedure LISTHD2()
V:=popSem(3)
if errorFound:=anyError(V) then return pushSem(errorFound)
pushSem(V[1]+1)
return
end

procedure LISTHD1()
V:=popSem(1)
if errorFound:=anyError(V) then return pushSem(errorFound)
pushSem(0)
return
end

procedure LISTN2()
V:=popSem(3)
if errorFound:=anyError(V) then return pushSem(errorFound)
put(P, [")",V[1]+1] )
pushSem(&null)
return
end

procedure LISTN1()
V:=popSem(2)
if errorFound:=anyError(V) then return pushSem(errorFound)
put(P, [")",V[1]] )
pushSem(&null)
return
end

procedure PROCFORDECL()
V:=popSem(3)
if errorFound:=anyError(V) then return pushSem(errorFound)
pushSem(V[1])
return
end

procedure PROCHD()
V:=popSem(1)
if errorFound:=anyError(V) then return pushSem(errorFound)
bn +:= 1; on := 0
put(P, ["proc",&null] )
```

```
pushSem(*P)
n +:= 1
N[n] := ["",m]
m := n
return
end

procedure PROCDEF()
V:=popSem(3)
if errorFound:=anyError(V) then return pushSem(errorFound)
put(P, ["endproc"] )
P[V[1]][2] := *P+1
bn -:= 1
n := m-1
m := N[m][2]
pushSem(&null)
return
end

procedure VALUE()
V:=popSem(1)
if errorFound:=anyError(V) then return pushSem(errorFound)
put(P, ["value"] )
pushSem(&null)
return
end

procedure CALL()
V:=popSem(2)
if errorFound:=anyError(V) then return pushSem(errorFound)
put(P, ["call"] )
pushSem(&null)
return
end

procedure LOADLOGVAL()
V:=popSem(1)
if errorFound:=anyError(V) then return pushSem(errorFound)
put(P, ["logval",V[1]] )
pushSem(&null)
return
end

procedure LOADNUM()
V:=popSem(1)
if errorFound:=anyError(V) then return pushSem(errorFound)
put(P, ["number",numeric(V[1].body)] )
pushSem(&null)
return
end

procedure LOADSYMB()
V:=popSem(1)
if errorFound:=anyError(V) then return pushSem(errorFound)
put(P, ["symbol",V[1].body] )
pushSem(&null)
return
end

```

```
procedure LOADUNDEF()
put(P, ["undef"] )
return
end

procedure PARENS()
V:=popSem(3)
if errorFound:=anyError(V) then return pushSem(errorFound)
pushSem(&null)
return
end

procedure INPUT()
put(P, ["in"] )
return
end

procedure UOP()
V:=popSem(2)
if errorFound:=anyError(V) then return pushSem(errorFound)
put(P, [V[1].body] )
pushSem(&null)
return
end

procedure BOP()
V:=popSem(3)
if errorFound:=anyError(V) then return pushSem(errorFound)
put(P, [V[2].body] )
pushSem(&null)
return
end

procedure UPLUS()
V:=popSem(2)
if errorFound:=anyError(V) then return pushSem(errorFound)
pushSem(&null)
return
end

procedure NEG()
V:=popSem(2)
if errorFound:=anyError(V) then return pushSem(errorFound)
put(P, ["neg"] )
pushSem(&null)
return
end

procedure CONJHD()
V:=popSem(2)
if errorFound:=anyError(V) then return pushSem(errorFound)
put(P, ["and",&null] )
pushSem(*P)
return
end

procedure CONJ()
V:=popSem(2)
if errorFound:=anyError(V) then return pushSem(errorFound)
```

```
P[V[1]][2] := *P+1
pushSem(&null)
return
end

procedure DISJHD()
V:=popSem(2)
if errorFound:=anyError(V) then return pushSem(errorFound)
put(P, ["or",&null] )
pushSem(*P)
return
end

procedure DISJ()
V:=popSem(2)
if errorFound:=anyError(V) then return pushSem(errorFound)
P[V[1]][2] := *P+1
pushSem(&null)
return
end

procedure TRUEPT()
V:=popSem(2)
if errorFound:=anyError(V) then return pushSem(errorFound)
put(P, ["else",&null] )
pushSem(*P)
return
end

procedure IFCLSE()
V:=popSem(3)
if errorFound:=anyError(V) then return pushSem(errorFound)
put(P, ["then",&null] )
pushSem(*P)
return
end

procedure IFEXPR()
V:=popSem(3)
if errorFound:=anyError(V) then return pushSem(errorFound)
P[V[1]][2] := V[2]+1
P[V[2]][2] := *P+1
pushSem(&null)
return
end

procedure LABSTMT()
V:=popSem(2)
if errorFound:=anyError(V) then return pushSem(errorFound)
pushSem(&null)
return
end

procedure LABDEF()
local t,s
V:=popSem(2)
if errorFound:=anyError(V) then return pushSem(errorFound)
t:=n
repeat {# write(N[t][1]," : ",V[1].body)
```

```
 if t<=m then
   return pushCTError("undeclared label "||V[1].body)
 if N[t][1]===V[1].body then break
 t -:= 1
}
if N[t][4]~===&null then
 return pushCTError("redefinition of label "||V[1].body)
s := N[t][3]
N[t][3] := *P+1
while s ~=== &null do {
 t := P[s][2]
 P[s][2] := *P+1
 s := t
}
pushSem(&null)
return
end

procedure BEGIN()
V:=popSem(1)
if errorFound:=anyError(V) then return pushSem(errorFound)
bn +:= 1
on := 0
put(P, ["begin"] )
n +:= 1
N[n] := ["",m]
m := n
pushSem(&null)
return
end

procedure BLKHD()
V:=popSem(3)
if errorFound:=anyError(V) then return pushSem(errorFound)
pushSem(&null)
return
end

procedure BLKBODY()
V:=popSem(3)
if errorFound:=anyError(V) then return pushSem(errorFound)
put(P, [";"] )
pushSem(&null)
return
end

procedure BLK()
V:=popSem(3)
if errorFound:=anyError(V) then return pushSem(errorFound)
put(P, ["end"] )
n := m-1
m := N[m][2]
bn := bn-1
pushSem(&null)
return
end
```

Chapter 5 Exercises

5.1 Change the exponentiation operator

Change EULER’s exponentiation operator from "**" to "^".

5.2 New unary operators

Implement two new unary operators for EULER:

- explode s, where s is a symbol, will yield a list of single character symbols which are the characters in symbol s.
- implode L, where L is a list of symbols, will yield a symbol which is the concatenation of the symbols in list L.

For example,

explode "frog"

yields ("f","r","o","g")

implode ("to","a","d")

yields "toad"

implode explode "frog"

yields "frog"

explode implode ("to","a","d")

yields ("t","o","a","d")

Note that the syntax allows several explodes and implodes to be used together and to operate on any primary.

Hint on implementation: You will need to change the scanner, the syntax (and regenerate the parser), the interpreter, and maybe the semantics routines. In short, you must make coordinated changes throughout the EULER compiler.

5.3 Change the symbol table

Change the symbol table in the EULER compiler to use a stack of Icon tables.

5.4 Use relative block numbers

Observe that the EULER implementation keeps around block numbers when they are not needed. The first field of an activation record contains the block number, which indicates the depth of nesting of the block. When the `@` instruction searches for a variable or formal parameter, it compares the block number of the activation record it is looking at with the block number desired (see procedure *reference,* lines 8-17 in the interpreter). Block number *j* is always nested within block *j-1*. When the `@` instruction is generated, the compiler knows the number of the block the instruction is in and the number of the block the variable is in, and hence how many levels back on the static chain procedure *reference* will travel before finding the variable.

Given this insight, make the following changes:

`@  on,levels`	Change the reference instruction to indicate the number of levels back along the static chain the variable or formal parameter is located.
`label  pa,levels`	Make the same change to the label instruction.

5.5 Peephole optimization

Peephole optimization is an improvement of generated code that replaces short sequences of instructions with shorter sequences. Perform at least the two following peephole optimizations:

replace	with	explanation
@ on,levels value	loadvalue on,levels	`loadvalue` is a new instruction that performs the combined operations of the two instructions it replaces. Most machines have both load and load-address instructions.
label pa,levels value	label pa,levels	The `value` instruction leaves a ProgRef value on the stack unmodified.

See if there are some other instruction sequences you can recognize and optimize.

5.6 Jump optimization

Jump optimization attempts to optimize collections of jump instructions. Since the names of jump instructions in the EULER abstract machine are based on the EULER constructions they are generated from, rather than on their behavior, it would be confusing to try to discuss the optimizations using their own names. We will describe some jump optimizations using the names given in the following table:

instruction	EULER name	mnemonic	description
`pajf` *`dst`*	`then` *`dst`*	pop and jump false	Pop the top value off the stack. If the value popped was *false*, jump to *dst*.
`pajt` *`dst`*	`----`	pop and jump true	Pop the top value off the stack. If the value popped was *true*, jump to *dst*.
`jfop` *`dst`*	`and` *`dst`*	jump false or pop	If the top value on the stack is *false*, jump to *dst*; otherwise pop it off the stack.
`jtop` *`dst`*	`or` *`dst`*	jump true or pop	If the top value on the stack is *true*, jump to *dst*; otherwise pop it off the stack.
`j` *`dst`*	`else` *`dst`*	jump	Unconditionally jump to *dst*.

Here are some examples of jump optimizations:

original instructions	replacement	similarly for instructions

`pajf L1`
`...`
`L1: j L2`
	`pajf L2`
	`...`
	`L1: j L2`
		`pajt L1,`
		`jtop L1,`
		`jfop L1, or`
		`j L1`

`jtop L1`
`...`
`L1: jtop L2`
	`jtop L2`
	`...`
	`L1: jtop L2`
		`jfop/jfop`

original instructions	replacement	similarly for instructions

`jtop L1`
`...`
`L1: jfop L2`
	`pajt L1+1`
	`...`
	`L1: jfop L2`
	`L1+1:`
		`jfop/jtop`

Implement at least these jump optimizations.

5.7 Add a while-expression.

Add a **while**-expression and accompanying **next-** and **break**-expressions.

5.7.1 *Syntax*

expr → **while** expr **do** expr

expr → **next**

expr → **break**

5.7.2 *Semantics*

expr → **while** expr_1 **do** expr_2

As usual, the while-expression repeatedly evaluates expression expr_2 as long as expression expr_1 evaluates to true. When expr_1 evaluates to false, the while-expresion terminates.

Since the while-expression is an expression, it must return a value. It returns a value of **false** if it is exited normally (by expr_1 evaluating to **false**) and the value **true** if it is exited via a break-expression.

expr → **next**

The next-expression will restart the while-expression from the beginning. The next-expression can be evaluated in either expr_1 or expr_2.

expr → **break**

The break-expression will make the enclosing while expression terminate and yield the value **true**. The break-expression can be evaluated in either expr_1 or expr_2.

5.7.3 Hints on implementation

5.7.3.1 Suggested translation:

source	translation
while e1 do e2	

```
         begin
Lnext:
         <e1>
         and       Lbreak
         <e2>
         ;
         else      Lnext
Lbreak:
         end
```

source	translation
next	

```
          popto     bn
          else      Lnext
```

source	translation
break	

```
          popto     bn
          logval    true
          else      Lbreak
```

The only new instruction is `popto` which removes all of the stack back to that activation record along the static chain which has block number *bn*. The `and` instruction is a conditional jump and the `else` instruction is an unconditional jump.

5.7.3.2 Suggested compiler data structures:

Keep a stack with one element for every enclosing while-expression. Each element of the stack contains three things:

1. The block number of the block created by the while-expression. This is the *bn* used in the `popto` instructions.
2. The address (position in P) of the Lnext label for the while-expression.
3. A linked list of `and` and `else` instructions jumping to the Lbreak label. These will be filled in at the end of the loop, when the position of the Lbreak label is known.