The Picky programming language v2.0

4/4/14

Francisco J Ballesteros
Gorka Guardiola Múzquiz
Laboratorio de Sistemas
Universidad Rey Juan Carlos

ABSTRACT

Picky is a programming language designed for use in a first level, introductory, programming course. The language is small and simple, and is strict regarding what is a legal program. This document describes the second version of the language. This new version has been reimplemented in go and has new multimedia facilities. It enables the programer to create a graphical user interface using the web browser with images and sound.

1. Motivation

Ada could be a good language for teaching, but it is quite verbose and utterly complex. This makes things hard for students in introductory courses, because there are many different constructs to master. Picking a subset is not doable in practice, because many features left out still show up even for modest subsets. Type safety is a must, but automatic features (like automatic dereferencing of pointers) makes it unclear for students what the code actually does. Also, control structures requiring *exit when* constructs are easily misused. File handling in Ada is clumsy, to say the least. For example, calling *End_Of_File* may block a program, reading from a terminal, and students will not know why. Furthermore, we teach that functions should not have lateral effects, but many file I/O tools are functions.

Low level languages, like C, are not suitable at all. Type safety is a must and structured data including strong typing and range checks are good to have when learning how to program for a first time.

Scripting languages do not enforce good practice, and have undesirable features in many cases. For example, including white space as part of the syntax (e.g., tabulators) or automatic declaration of variables.

Object oriented languages are too complex for use as a first language. They may be popular, but they are not clean and look like magic to most students.

Pascal is a good first language. However, its control syntax is verbose. Also, the language syntax is more complex than needed. For example, the use of semicolons as separators instead of terminators for sentences is a problem for students. They end up guessing when to add a semicolon and when not to add one.

We wanted a language as simple as Pascal, with terse syntax (like C), and a realistic handling of file I/O. File I/O is important not just to perform I/O, but also to make students learn how to use control structures to guide data consumption without violating file I/O rules imposed by the file abstraction. As a result, we designed a new language, called Picky.

The language compiles to byte-code for an abstract machine called PAM. An interpreter for PAM code is supplied along with the compiler. This isolates students from portability issues that would arise otherwise.

When a kid learns how to ride a bicycle it is convenient to use side-wheels for a while. Only after such artifact is under control, a new bicycle (one without side-wheels, and perhaps with an engine) is more convenient. In the same way, Picky is highly restrictive regarding what can be done and what can not in a program. It has side-wheels attached. Both the compiler and the run time include extra checks and waste memory and time to provide additional safety features (e.g., more informative diagnostics regarding accidental use of dangling pointers).

2. The language

2.1. Picky programs

Picky has control structures reminiscent of C and data declarations in the style of Pascal. A source program is made of a single file. This is a hello world:

```
1	/*
2	 *   Hello world
3	 */

5	program Hello;

7	procedure main()
8	{
9		writeln("hello, world");
10	}
```

Comment syntax is taken from C. A program is introduced by a *program* clause (line 5) that assigns an identifier to the program. A program may have constant and type definitions, variable declarations, procedure definitions and function definitions. A procedure named *main* must be included, like in C. The program starts executing its body and terminates when returning from it.

All declarations and statements are terminated by a semicolon, but note that procedure and function definitions are not terminated by a semicolon. Constants, types, procedures, and functions may not be declared within the scope of a procedure or function. That is, subprograms may not be nested and constants and types must be declared in the global scope.

The language is case-sensitive. Thus, *main*, *Main*, and *MAIN* are different identifiers. An identifier must start with an alpha rune followed by zero or more alphanumeric runes.

The following names are reserved and correspond to keywords, pre-defined variables, types, procedures, functions, and constants. All other names are available for new identifiers.

acos	ENote	gfillrgb	Minint	Sheep
and	Eof	gkeypress	Minstrength	Shift
ANote	Eol	gline	new	sin
array	Esc	gloc	nil	sleep
AsharpNote	exp	GNote	NoBut	sqrt
asin	Fail	gopen	not	stack
atan	False	gpencol	Nul	stdin
Beep	fatal	gpenrgb	of	stdout
Black	feof	gpenwidth	Opaque	stdgraph
Blue	feol	gplay	open	succ
BNote	fflush	gpolygon	or	switch
Bomb	flush	greadmouse	Orange	Tab
case	FNote	Green	peek	Tada
close	for	GsharpNote	Phaser	tan
CNote	fpeek	gshowcursor	pow	Tlucid
consts	fread	gstop	pred	Transp
cos	freadeol	gtextheight	procedure	True
CsharpNote	freadln	if	program	types
Ctrl	frewind	Left	rand	Up
data	FsharpNote	len	read	vars
default	function	log	readeol	while
Del	fwrite	log10	readln	White
dispose	fwriteeol	Maxchar	record	Woosh
DNote	fwriteln	Maxint	Red	write
do	gclear	Maxstrength	ref	writeeol
Down	gclose	MetaLeft	return	writeln
DsharpNote	gellipse	MetaRight	Right	Yellow
else	gfillcol	Minchar	Rocket	

A program starts with the *program* clause and must include a procedure with no parameters and named *main*, as shown.

A program may also include one or more constant declaration blocks, one or more type declaration blocks, one or more variable declaration blocks, and procedure and function definitions. The scope for a declaration goes from the point where it happens in the source to the end of file.

Constant, type, and variables declaration blocks start with the keyword *consts*, *types*, and *vars* (respectively) followed by declarations. This program is an example:

```
program Xample;

consts:
     C1  = 11;
     Greet = "hi";

types:
     Tmonth = (Ene, Feb, Mar);
     Tyesno = bool;

consts:
     Zmonth = Ene;

vars:
     a: month;
```

```
procedure main()
{
	/* ... */ ;
}
```

2.2. Constants

Constants are defined like in the example. Constants for basic types have data types derived from their values, which may be expressions as long as their resulting value may be computed at compile time.

Integer literals are digits, base 10, one after another. A leading plus or minus sign is actually an unary expression adjusting the sign of the following operand. Float (real) literals are digits with a decimal point and at least one more digit, perhaps followed by an exponential notation (i.e., an ‘‘*E*’’ an optional sign, and one or more digits). Boolean values are named *True* and *False*. Character literals are a single rune within single quotes. Array of character (string) literals are one or more runes within double quotes. These are some examples:

```
consts:
	C1 = 11;	/* int */
	C2 = −2;	/* int */
	C3 = 3.0;	/* float */
	C4 = 4.3E10;	/* float */
	Ok = True;	/* bool */
	X = 'X';	/* char */
	Msg = "hi";	/* array[0..1] of char */
```

Aggregates are discussed later, along with arrays and records.

2.3. Basic data types

Picky is strongly typed. Too strongly, hence its name. Basic types are *bool*, *char*, *int*, *float*, and *file*. They correspond to booleans, characters, integers, real numbers in floating point and external (text) files.

Two types are compatible (for assignment and other operators) only if they have the same name. Predefined types also obey this rule. Constants and literals are an exception, they belong to ‘‘universal’’ types that are assumed to be compatible with any basic data type of the same kind. This is reasonable, for example, to permit using integer literals in expressions that belong to a user defined integer type. Another exception are subranges. Subranges do not introduce a new type; they declare a restriction defining a subset of an existing type.

A type definition defines a new type and declares its name. For example

```
types:
	Apples = int;
	Oranges = int;
```

defines two new types: *Apples* and *Oranges*. It is not legal to mix apples with oranges, and it is not legal to mix any of them with *int* values. However, integer constants and literals may be mixed with any of them.

Picky also defines three builtin types, *button*, *strength*, and *opacity*, used for mouse buttons, color strength and color opacity respectively. The *opacity* type derives from *float* and can be any value from 0.0 to 1.0. The type *strength* derives from *int* and can be any value from 0 to 255. The type *button* also derives from int and can take any positive value depending on the number of buttons in the mouse. All of them follow the same rules as user defined types and are incompatible with the supertype but compatible with constants and literals.

2.4. Predefined variables and constants

There are several constant character values defined:

Operator	Meaning
`Eof`	End of file
`Eol`	End of line
`Tab`	Tabulator
`Esc`	Escape key
`Nul`	Null byte
`Ctrl`	Control key
`Del`	Del key
`Down`	Down arrow key
`Left`	Left arrow key
`MetaLeft`	Meta left key
`MetaRight`	Meta right key
`Return`	Return key
`Right`	Right arrow key
`Shift`	Shift key
`Up`	Up key

Note that *Return* and *Eol* are represent diferent things. The first one is used when reading which keys are pressed from a graphical user interface in a non–blocking fashion. The second one represents end of line in a portable fashion.

Constants *Maxint* and *Minint* report the maximum and minimum values for the *int* data type. Like *Maxchar* and *Minchar* do for the *char* data type. Three constants for values of opacity are defined, *Opaque*, *Transp, Tlucid*. There is an enumeration of colors defined in picky: *Black*, *Red*, *Green*, *Blue*, *Yellow*, *Orange*, and White. A constant of type *button,* used to report that no button is pressed is defined: *NoBut*. An enumeration of sounds is also defined: *Woosh*, *Beep*, *Sheep*, *Phaser*, *Rocket*, *ANote*, *AsharpNote*, *BNote*, *CNote*, *CsharpNote*, *DNote*, *DsharpNote*, and *ENote*.

Predefined variables named *stdin*, *stdout*, and *stdgraph*, of type *file*, exist for standard input and output and graphics.

The special value *nil* is predefined and represents a null pointer. It is type compatible with any pointer type.

2.5. Operators and builtin operations

We describe here the operators available in the language (but for the `len` operator, which is discussed along with structured data types). For binary operators, both operands must be type compatible. The resulting type is always of the same type of the arguments, but for obvious exceptions (i.e., relational operators always yield *bool* values).

Values of data types other than *file* may be compared using equality operators:

Operator	Meaning
==	Equal to
!=	Not equal to

Equality yields *True* if and only if values are equal. Inequality yields *True* if and only if values are not equal. For structured types (described later), these operators compare their inner elements, one by one.

Values of ordinal data types (that is, *bool*, *char*, *int*, and user defined enumerations) have fixed positions in their abstract sets, and may be compared using the following:

Operator	Meaning
<	Less than
>	Greater than
<=	Less or equal than
>=	Greater or equal than

Ordinal values have two more functions defined:

Built−in	Meaning
`pred(v)`	Predecessor of v
`succ(v)`	Successor of v

Pred yields the predecessor of *v* in the data type. *Succ* yields the successor of *v* in the data type.

Boolean values accept usual boolean operators:

Operator	Meaning
`and`	binary logical and
`or`	binary logical or
`not`	unary logical negation

And and *or* evaluate both operands. That is, there is no short−circuit evaluation as found in C.

Numeric data types accept the following operators, their operands must be type compatible, as usual. Not all operators are defined for both integers and floating point numbers (the table shows legal operand types).

Operator	Meaning	Argument types
+	binary addition or unary nop	float int
−	binary subtraction or unary sign change	float int
*	binary multiplication	float int
/	binary division	float int
%	binary modulus	int
**	binary exponentiation	float int

Expressions may be parenthesized as required. The precedence of operators is indicated by the following table, from low to high precedence. Operators in the same row have the same precedence. All operators associate to the left. Expressions are evaluated left to right.

	Precedence
	`or and`
	`== != < > <= >=`
low	+ − *(binary)*
	* / %
	**
high	+ − *(unary)*
	`len not`

The *len* operator returns the number of elements in the object given as an argument. It is discussed later, in the section for structured types.

The following functions are defined for *float* arguments, and yield a *float* result. They inherit their names and behavior from C, so we do not describe them any further.

Function Meaning	
`acos(r)`	arc–cosine
`asin(r)`	arc–sine
`atan(r)`	arc–tangent
`cos(r)`	cosine
`exp(r)`	exponential
`log(r)`	logarithm
`log10(r)`	base 10 logarithm
`pow(r1, r2)`	power
`sin(r)`	sine
`sqrt(r)`	square root
`tan(r)`	tangent

The following functions and procedures are defined to perform I/O. Some of them operate on *stdin* or *stdout*, others operate on the file given, as indicated. The argument *obj* may be a value or l–value of any basic type (i.e., non structured type), and it may be also an *array* of *char*.

Built–in	**Proc/Func**	**Meaning**
`close(file)`	procedure	Close the file
`eof()`	function	Report if Eof has been met in stdin
`eol()`	function	Report if Eol has been met in stdin
`feof(file)`	function	Report if Eof has been met in file
`feol(file)`	function	Report if Eol has been met in file
`fflush(file)`	procedure	Flush the output buffer for file
`flush()`	procedure	Flush the output buffer for stdout
`fpeek(file, char)`	procedure	Look ahead next char from file, or Eof, or Eol
`fread(file, obj)`	procedure	Read object from text representation in file
`freadln(file, obj)`	procedure	Idem, and skip the rest of line (and Eol)
`freadeol(file)`	procedure	Read end of line from file
`frewind(file)`	procedure	Seek to start of file
`fwrite(file, obj)`	procedure	Write text representation for object in file
`fwriteln(file, obj)`	procedure	fwrite(file,obj); fwriteeol(file);
`fwriteeol(file)`	procedure	Write end of line in file
`open(file, name, mode)`	procedure	Open file with given name for mode (which may be "r", "w", or "rw")
`peek(char)`	procedure	Look ahead next char from stdin, or Eof, or Eol
`rand(nmax, r)`	procedure	generate a number [0, nmax)
`read(obj)`	procedure	Read object from text representation in stdin
`readln(obj)`	procedure	Idem, and skip the rest of line (and Eol)
`readeol()`	procedure	Read end of line from stdin
`sleep(n)`	procedure	Suspend execution for n milliseconds

`write(obj)`	procedure	Write text representation for object in stdout
`writeln(obj)`	procedure	write(obj); writeeol();
`writeeol()`	procedure	Write end of line in stdout

The following functions and procedures perform I/O on a file representing a user interface (the ones described above also work on this kind of files, but not the other way round), passed as a first argument. The user interface can be used with the blocking interface as a regular file and text will be drawn at the pen's position. When reading, it behaves as a regular blocking file, similar to *stdio* (except not all keys are reported by the UI). The end of the connection with the UI is signaled by *Eof*. Once *Eof* is reported (be it through a read or while reading the pressed keys), using the UI results in an error. Once a keypress is reported, that keypress is consumed and will not be reported when performing a read.

All the drawing routines change the buffer and when a flush is performed the changes are seen on the UI. The coordinates for UI are integers and in (smaller than real) virtual pixels.

Built−in	**Proc/Func**	**Meaning**
`gclear(g)`	procedure	Cleans the buffer, resets the text position
`gclose(g)`	procedure	Closes the interface
`gellipse(g, x, y, r1, r2, α)`	procedure	Draws an ellipse with angle
`gfillcol(g, c, op)`	procedure	Sets the fill colo with opacity
`gfillrgb(g, rs, gs, bs, op)`	procedure	Sets the fill RGB color with opacity
`gkeypress(g, k)`	procedure	Reads keys pressed, k is a char or array of char
`gline(g, x1, y1, x2, y2)`	procedure	Draws a line
`gloc(g, x, y, α)`	procedure	Sets text pen position and angle
`gopen(g, name)`	procedure	Opens a new UI, naming it
`gpencol(g, c, op)`	procedure	Sets the pen color with opacity
`gpenrgb(g, rs, gs, bs, op)`	procedure	Sets the pen RGB color with opacity
`gpenwidth(g, w)`	procedure	Sets the pen width
`gplay(g, s)`	procedure	Plays a sound
`gpolygon(g, x, y, r, nsides, α)`	procedure	Draws a polygon
`greadmouse(g, x, y, b)`	procedure	Reads mouse position and button
`gshowcursor(g, isvis)`	procedure	Shows the mouse cursor
`gstop(g)`	procedure	Stops any sound being played
`gtextheight(g)`	function	Returns current text height

L−values of pointer types may use the following builtins to allocate and deallocate memory.

Built−in	**Proc/Func**	**Meaning**
`dispose(ptr)`	procedure	Dispose memory referenced by ptr
`new(ptr)`	procedure	Set ptr to point to newly allocated memory

Three other built−ins are provided for debugging and abnormal termination.

Built−in	**Proc/Func**	**Meaning**
`fatal(text)`	procedure	Print text and abort execution
`stack()`	procedure	Dump the stack for debugging

`data()`	procedure	Dump global data for debugging

2.6. Type casts

In general, the language does not permit type casts. However, type casts are permitted to convert ordinals to the integer representing their position in the type and vice-versa. Also, integers may be converted to floating point numbers and vice-versa.

To convert a value to a type use the target type name as a function. For example, these are legal expressions:

```
char(int('A') + 1)
float(3)
int(4.2)
```

2.7. Basic type definitions

A new type may be defined as new instance of an existing type by using the existing type as its definition. For example,

```
types:
     Apples = int;
     Oranges = int;
```

Enumerated types are also ordinal types, and are defined by enumeration of their literals as in the example:

```
1    types:
2         Month = (Jan, Feb, Mar);
3         Yesno = (No, Yes);
```

Line 2 introduces both the *Month* data type and new literals *Jan*, *Feb*, and *Mar*.

Subranges of existing ordinal data types (i.e., *bool*, *char*, *int*, and enumerated data types) may be declared. Subranges do not introduce a new data type. They introduce a range limit for an existing type, and remain type compatible with that type. Ranges are checked at run-time and may lead to a program panic if not obeyed by the user code. A subrange is defined by naming the actual type and the range, as in this example:

```
types:
     Mrange = Month Jan..Feb;
     Letter = char 'a'..'z';
```

2.8. Structured Types

Array types may be declared using an ordinal type (usually a subrange) as an index specifier and any other type as the element specifier. For example:

```
types:
     Days = array[Month] of int;
     Days2 = array[Jan..Feb] of int;
```

There is no data type for strings. Instead, an array of characters indexed by integers starting with 0 is used.

The syntax does not allow to nest definitions for data types. Only in the range index specifier can be nested, instead of defining a type name and then using it. This enforces the policy of declaring type names for inner components of structured data. As a result, multi-dimensional arrays require defining the type for a row or column (in *n*-1 dimensions) and then the type for the array, using the previous one as the element type. Syntax to refer to array elements is as expected in C-like languages:

```
days[Jan]
matrix[3][2]
```

Record (or structure, or tuple) types may be declared using the *record* keyword and a bracketed list of field declarations. As in this example:

```
program Example;
types:
     Prange = int 1..10;
     Point = record
     {
          x: int;
          y: int;
     };
     Points = array[Prange] of Point;
     Poly = record
     {
          points: Points;
          npoints: int;
     };
```

It is feasible to switch on a value of a enumerated-type field to define some fields only for particular values of that switch-field. For example:

```
Cmd = record
{
     code: Code;
     kind: Kind;
     switch(kind){
     case Rangecmd:
          r: Rangetype;
     case Recmd, Strcmd:
          s: Str;
     case Intcmd:
          i: int;
     }
};
```

In this case, the field *s* is available only when the field *kind* has either *Recmd* or *Strcmd* as values. For values of *kind* other than *Rangecmd*, *Recmd*, *Strcmd*, and *Intcmd*, the only fields of *Cmd* are: *code* and *kind*.

As explained before, type definitions may not be nested. For example, it is imperative to define the types *Point* and *Points* in this example before defining *Poly*. Otherwise, members of *Poly* couldn't be arrays or records. Only *Prange* might be avoided, by using the range directly in the definition of *Points*.

Syntax for member access is as expected, using the dot notation. For example:

```
poly.points[1].x
```

The operator `len` may be used with a type, variable, or constant name to yield the number of members of the given object or type. For example,

```
len Points
```

would be the integer value `10` in the previous example. This operator is evaluated always at compile time and does not evaluate its arguments.

2.9. Aggregates

For arrays and records, literal values may be constructed using the type name as a (constructor) function and supplying as arguments values of appropriate types for each one of the members, in the order used in the type definition. An aggregate value may be used in any place a value of the corresponding type may be used, including constant definition and subprogram arguments. For example:

```
types:
     Arry = array[0..1] of char;
     Word = record{
          chars: Arry;
          n: int;
     };

consts:
     Greet = Word("hi", 2);
```

2.10. Pointers

A pointer data type refers to another type and permits using *new* and *dispose* to handle dynamic variables of the pointed-to type. Type definition uses the ‘‘^’’ notation, taken from Pascal:

```
1    types:
2         Arry = array[1..10] of int;
3         Iptr =  ^int;
4         Aptr = ^arry;
```

Line 2 declares an array data type used in line 4, to declare a pointer to *Array* data type. Line 3 declares a pointer to integer. It is legal to declare a pointer to a type that is not yet defined in the program, but the target type must de defined later. This permits declaring circular data types, like linked lists. In no other case may a type be defined in terms of not yet defined types.

Syntax to dereference a pointer value is taken from Pascal, and also uses the ‘‘^’’ sign:

```
iptr^ = 2;
aptr^[1] = iptr^;
```

All memory allocated with *new* must be released by calling *dispose* before completion of the program, or the program will abort and report memory leaks.

2.11. Procedures and functions

Procedures are actions with names and do not return values. Argument passing is by-value by default. Multiple arguments are declared separated by commas. Using the keyword *ref* before an argument name makes pass-by-reference active for that parameter. For example,

```
procedure initword(ref w: Tword)
{
     w = nil;
}
```

defines a procedure with a single argument, passed by reference, of type *Tword*. Instead,

```
procedure addtoword(ref w: Tword, c: char)
{
     ...
}
```

defines a procedure with two arguments. *w* is of type *Tword* and passed by reference. However, *c* is of type *char* and is passed by value.

Functions are declared in a similar way, using the *function* keyword and declaring the return type like in this example:

```
function isblank(c: char): bool
{
     return c == ' ' or c == Tab or c == Eol;
}
```

All function arguments must be passed by value. All in all, we teach that functions should have no lateral effects and should preserve referential transparency.

2.12. Global and local variables

Global variables are declared like types and constants, with a declaration block. In this case, the keyword *vars* must be used instead. For example:

```
program Xample;
vars:
     n: int;
procedure main()
{
     ...
}
```

The declaration uses the pascal colon syntax. Unlike in Pascal, it is not allowed to declare a type on the fly in the variable declaration. A type identifier is required after the colon. Also, there is no initialization syntax, by design. Variable initialization must happen in the body of procedures and functions.

All variables are initialized to random values. That means that it is unlikely to find them zeroed even the first time they are used.

Local variables are declared within the procedure or function header and its body. In this case, the *vars* declaration specifier is not used. Procedures and functions may not contain constant or type definitions and so, declarations always refer to (local) variables.

This example declares a local variable named *f*:

```
function fact(n: int): int
     f: int;
{
     ...
     return f;
}
```

2.13. Statements

Statements are not expressions (like in C), but actions (like in Pascal). They must be terminated by a ‘‘;’’. The null statement is just the ‘‘;’’, on its own. Statement blocks are enclosed by curly brackets, as it has been seen for procedure and function bodies, which are blocks.

Assignment uses the ‘‘=’’ operator, like in C. For example:

```
x = 0;
```

Needless to say that arguments must be type compatible and that the left part must be an L–value.

Function calls are not allowed as statements, because they are expressions. Procedure calls are allowed as statements (and not in expressions), and use the obvious syntax:

```
write(3);
writeln();
fwrite(stdout, Eol);
```

If there are no arguments, parenthesis must still be supplied.

The statement *return* returns a value from a function, like in the example of the previous section. It is required that *return* is the last statement in the function body. Early returns are not allowed. It is permitted to use a conditional as the last statement in a function, as long as all its arms include a *return* statement as their last sentence. Procedures may not use *return*.

2.14. Control structures.

Conditional execution is controlled by the *if* statement, which borrows syntax from C. But there are differences. Statements used for *then* and *else* arms must be blocks. That is, brackets must be used always. For example:

```
if(len(w) > len(max)){
     max = w;
}
```

or

```
if(c == ' ' or c == '	'){
     read(c);
}else if(c == Eol){
     readeol();
}
```

Multiple *if* statements may be chained by using an *if* statement directly in the *else* of a previous *if*.

```
if(c == ' ' or c == '	'){
     read(c);
}else if(c == Eol){
     readeol();
}
```

while and *do–while* loops borrow the syntax from C:

```
do{
     read(c);
}while(not eof() and isblank(c));
```

and

```
while(w != nil){
     tot = tot + w^.len;
     w = w^.next;
}
```

The *for* loop reminds to that of C, but has semantics closer to Pascal. Two expressions, an initialization and a condition, are present within parenthesis in the loop header. The initialization must be an assignment for a variable of an ordinal type. The condition must use any of the ‘‘<’’, ‘‘<=’’, ‘‘>’’, ‘‘>=’’ operators. The first two ones make the variable increase automatically after each iteration. The last two ones make the variable decrease automatically after each iteration. For example:

```
for(i = 0, i < Nitems){
	write(item[i]);
}
```

After the *for* loop, the control variable would be equal to the value on the right of the condition. This implies that there is no out of range condition for the control variable even when using ‘‘<=’’, or ‘‘>=’’ with the first or last valid value of an ordinal type. In our example, *i* value would be *Nitems* when the loop is done.

Multi−way conditionals use a *switch* syntax that reminds to (but differs from) that in C. Unlike in C, there is no fall−through; and there is no *break* statement. Expressions used in each *case* may be single values (of an ordinal type), or multiple values separated by commas (matching any of the arguments), or a range using the *dot−dot* notation. For example:

```
switch(4){
case 3,4..8:
	c = True;
case 1..4:
	c = True;
case 5:
	c = True;
default:
	;
}
```

3. The compiler

The picky compiler, *pick*, has been implemented in go. Ports to Linux, Windows and MacOS X are available. The description of the compiler provided in this section corresponds to an early version of the implementation. It is meant to provide a hint to people that must modify the compiler, but it is not up to date with respect to the implementation. The language description of previous sections is, of course, up to date.

The compiler is implemented using *go yacc*, and should be easy to understand. There are several things to know before attempting to modify it, which are documented here.

Symbol table handling as implemented is fast enough, but it is both simple and clumsy, and is the first thing that should be improved if more work is put in the compiler.

There are no warnings. All diagnostics correspond to compile time errors. In many cases, when an error is detected, a symbol or node in the syntax tree is still built, for safety; other parts of the compiler still get a data structure as expected, and it’s less likely that an invalid value causes a bug.

3.1. Symbol table

The symbol table is implemented as a stack of environments

```
/*
 * One per program, procedure, and function.
 * Used to keep symbols found in it and also to collect
 * definitions for arguments, constants, types, variables, and statements.
 */
type Env struct {
     id   uint
     tab  map[string]*Sym     // symbol table
     prev *Env      // in stack
     prog *Sym      // ongoing program, procedure, or function
     rec  *Type     // ongoing record definition
}
```

The global *env* points to the top of the stack. There is an initial environment used for the top-level (the outer scope). Another environment is pushed for each procedure, function, argument list, and record field list that is found. In some cases, the attributes in the grammar are not used to populate a node in the syntax tree. Instead, the global *env* is accessed to locate the procedure, function, or program being defined. The same is done to define fields for records. In most other cases, attributes as handled by *yacc* suffice.

Each environment is a map that keeps symbols for the compiler. Two additional maps are kept. One to store strings and another to store keywords.

```
var (
     strs = make(map[string]*Sym, Nbighash) // strings and names
     keys  = make(map[string]*Sym, Nhash)    // keywords and top-level
)
```

The former is used to keep an entry for each name found in the source. For simplicity, it maintains *Syms* and not strings. The later is used to keep keywords and global definitions. The scanner (done by hand) looks up in these tables to learn if a token for a keyword should be given to the parser. In most other cases, it allocates a new entry in the strings table and returns its symbol.

The grammar uses different tokens for identifiers and type identifiers. Therefore, the scanner checks if an (already defined) identifier is for a type or for any other value.

A symbol is represented by these data structures. For simplicity, the same data structures are used to correspond to nodes in the syntax tree for expressions, albeit strictly speaking they are not symbols.

```
type Val struct {
     //--one of:
     ival int

     rval float64

     sval string

     vals *List
}
```

```
// Symbol table entry.
type Sym struct {
     id     uint
     name   string
     stype  int
     op     int
     fname  string
     lineno int
     ttype  *Type

     //--one of:
     tok int

     Val

     used int
     set  int

     left  *Sym
     right *Sym     // binary, unary


     fsym  *Sym     // Sfcall
     fargs *List

     rec   *Sym     // "."
     field *Sym

     swfield *Sym   // switch field
     swval   *Sym   // variant
}
```

The struct(s) correspond to attributes for the symbol and backend information. In general, a symbol has a name, belongs to a type of symbol (*stype*) and depending on the type may correspond to one operation or another (*op*). These are the types of symbols known:

```
// symbol types and subtypes
const (
     Snone   = iota
     Skey    // keyword
     Sstr    // a string buffer
     Sconst  // constant or literal
     Stype   // type def
     Svar    // obj def

     Sunary  // unary expression
     Sbinary // binary expression
     Sproc   // procedure
     Sfunc   // function
     Sfcall  // procedure or function call
)
```

Symbols used to represent expressions carry in *op* the operation for the node:

```
const (
	// Operations besides any of < > =  + - * / % [ . and ^
	Onone = iota + 255
	Ole
	Oge
	Odotdot
	Oand

	Oor // 5 + 255
	Oeq
	One
	Opow
	Oint

	Onil // 10 + 255
	Ochar
	Oreal
	Ostr
	Otrue

	Ofalse // 15 + 255
	Onot
	Olit
	Ocast
	Oparm

	Orefparm // 20 + 255
	Olvar
	Ouminus
	Oaggr
)
```

In some cases, a symbol keeps a list of symbols as children. In all such cases, a *List* structure is used:

```
type List struct {
	kind int
	item []interface{}
}
```

where *kind* must be any of

```
const (
	// List kinds
	Lstmt = iota
	Lsym
)
```

For example, argument lists are lists of kind *Lsym*, and statement blocks are lists of kind *Lstmt*. The functions *addstmt*, *addsym* and methods *getsym*, and *getstmt* are used to manipulate lists conveniently.

An important symbol type is that for programs (and procedures and functions). It holds a *Prog* structure as its value, also linked from the corresponding *Env* structure.

```
type Prog struct {
	psym   *Sym
	parms  *List
	rtype  *Type // ret type or nil if none
	consts *List
	types  *List
```

```
     vars   *List
     procs  *List
     stmt   *Stmt
     b      *Builtin
     nrets  int


     // backend
     code   Code
     parmsz uint
     varsz  uint
}
```

The parser adds new symbols to the lists of constants, types, variables, and procedures/functions, as new elements are analyzed in the source. The single *stmt* is a block for the body of the procedure or function. For built-ins, *b* keeps a *Builtin* structure used to decorate the parser node with attributes and to encode the type signature.

```
type Builtin struct {
     name string
     id   uint32
     kind int
     args string
     r    rune
     fn   func(b *Builtin, args *List) *Sym
}
```

3.2. Data types

Each symbol is expected to have a *type* attached. The type is described by this data structure:

```
// Types
type Type struct {
     op     int
     sym    *Sym
     first  int
     last   int


     //--one of:
     lits   *List // Tenum

     ref    *Type // Tptr

     super  *Type // Trange

     idx    *Type // Tarry, Tstr
     elem   *Type

     fields *List // Trec

     parms  *List // Tproc, Tfunc
     rtype *Type

     //--
     // backend
     id    uint
     sz    uint
}
```

Type constructors allocate new structures. Two types are compatible if their address in memory are the same. Exceptions are made to support universally compatible data types, as used for constants.

The *op* field in *type* identifies the kind of type. It is any of:

```
const (
	// Type kinds
	Tundef = iota
	Tint
	Tbool
	Tchar
	Treal

	Tenum // 5
	Trange
	Tarry
	Trec
	Tptr

	Tfile // 10
	Tproc
	Tfunc
	Tprog
	Tfwd

	Tstr // 15; fake: array[int] of char; but universal
	Tstrength
	Topacity
	Tcolor
	Tbutton
	Tsound
	Tlast
)
```

Type *Tfwd* is used to temporarily define a type as a forward declaration. This is used for pointers, which permit the target type to be defined later. Type *Tstr* is an artifact, to represent strings which are type–compatible with arrays of characters of the same length.

All ordinal types have their first and last values stored in their *Type* structure. This is to perform range checks without paying attention to the difference between types and subtypes (only subranges as of today).

3.3. Statements

Statements are described by *stmt* structures:

```
type Stmt struct {
	op     int
	sfname string
	lineno int
```

```
    //--one of:
    list *List // '{'

    lval *Sym // =
    rval *Sym

    cond    *Sym // IF
    thenarm *Stmt
    elsearm *Stmt


    fcall *Sym // FCALL

    expr *Sym // RETURN, DO, WHILE, FOR, CASE
    stmt *Stmt
    incr *Stmt // last statement in fors (i++|i--)
}
```

The *op* field identifies the kind of statement. A token representative of the statement is used for this purpose. The union keeps the information describing the statement.

Statements for *for* loops are rewritten as a block that contains the initialization, a *while* loop, and its body adjusted to include the increment or decrement for the control variable.

Switch statements are also rewritten, to use a sequence of chained *if−then−else* statements, each one checking the value of the expression we are switching on. To prevent multiple evaluation of the *switch* expression, a variable is declared by the compiler for each such statement. The *switch* is rewritten to initialize the variable with the value of the expression, and then execute the chained *if* corresponding to the branches.

3.4. Builtins and predefined identifiers.

Builtin procedures and functions have type signatures generated from a description string within the front−end. Arguments are checked by a generic builtin type check function, which takes into account the polymorphic nature of procedures like *write*.

Builtin functions check to see if their arguments are evaluated as a result of constructing their nodes in the front−end. In that case, if the builtin may yield a value at compile time, the function call is replaced by the resulting value. The implementation tries to check if arguments are legal (e.g., would cause a floating point exception) and issue a sensible diagnostic otherwise. This process is guided by a *Builtin* structure as shown before.

Calls to file procedures and functions that operate on *stdin* and *stdout* are rewritten to pass the file explicitly, using the variants of the builtins that accept a file argument.

Pre−defined constants and variables are added to the environment for the top−level scope as soon as the parser tries to declare a program. Afterwards, they are handled like user defined objects.

3.5. Code generation

Code generation is straightforward, and uses back−patching to set label addresses. Procedure are called by procedure number, and not by procedure addresses. Therefore, this mechanism is not applied in this case.

Code is generated in blocks (one per procedure), using this structure:

```
// generated code
type Code struct {
     addr  uint32
     pcs   *Pcent
     pcstl *Pcent
     p     []uint32
     np    uint
     ap    uint
}
```

Here, *p* is the pointer to byte-codes (actually using a full *uint32* each); *np* is the number of byte-codes (words) produced, and *ap* is the number of byte-code slots (words) available in *p*.

For each statement, and for symbol and expression nodes, entries to match program counter to source file and line are linked into the *code* structure.

```
// pc/src table
type Pcent struct {
     next *Pcent
     st   *Stmt
     nd   *Sym
     pc   uint
}
```

Either *st* or *nd* is used, not both at the same time.

4. Error management

Panic is used in the compiler for fatal errors. Unexpected panics (i.e. those with "runtime error:" as a prefix, write an "internal error:" message. The presence of an "internal error:" message means that there is a bug in the compiler. The '-d' flag can be used in that case to dump the go stack. The same is true for the interpreter.

5. The interpreter

The description of the interpreter provided in this section corresponds to an early version of the implementation. It is meant to provide a hint to people that must modify the interpreter, but it is not up to date with respect to the implementation. The language description of early sections is, of course, up to date.

The interpreter, *pam*, implements an abstract machine known as PAM. The machine is a stack based machine. Most operations take arguments from the stack and replace them with a result, pushed also on the stack. There is a single flow of control, guided by an (almost) endless loop switching on the instruction type.

The interpreter leaks memory for storage allocated with *new*, (by keeping the references around so they are not garbage collected) to detect when disposed data structures are used and issue more descriptive diagnostics than ''segmentation violation''.

Also, it checks that assigned values are in range, more often than needed, to try to detect constraint errors early in the execution.

All memory, both data, stack variables, and dynamic memory, is initialized with random values, to let the user discover early that variable initialization is missing. Such random values are always odd, to recognize pointer values not initialized, and issue a descriptive diagnostic for that case at run time, instead of a ''segmentation violation'' or producing a heisen-bug.

5.1. PAM

PAM is the Picky Abstract Machine. It has the following elements:

- Some registers:

  pc Program counter. Addressing words, each one a byte-code.

  fp Frame pointer. Addressing bytes. To locate the activation frame for the current procedure.

  sp Stack pointer. Addressing bytes. To locate the top of the stack.

  vp (Local) Variable pointer. Used to translate local variable addresses into actual memory addresses.

  ap Argument pointer. Used to translate local argument addresses into actual memory addresses.

  pid Procedure identifier. Used to locate the descriptor for the procedure executing (or function).

- Text memory. Word addressed area of memory used to keep byte codes. Each byte code is a word, not a byte. Operations taking an argument use another word for the argument. The *pc* register indexes this memory, starting at 0.
- Stack memory. Byte addressed area of memory containing global variables (bottom of stack) and activation frames for procedures and functions. Stack addresses are machine addresses (i.e., actual addresses as used by the go implementation of PAM). All of *sp*, *fp*, *vp*, and *ap* point into this memory (i.e., they are integer indexes in the implementation).

  In order to simplify the implementation of the go interpreter, actual machine addresses are pushed into the stack. Slices are recovered in an unsafe way when poping from the stack. As pointer descriptors are kept around the garbage collector has a reference and does not free the referenced structures.

- Dynamic memory. Dynamic variables are stored using the underlying go heap. However, pointer values are references to descriptors that refer to the actual memory allocated. This is used as a fence to detect run time errors in user pointers, to issue diagnostics that help.
- Procedure descriptors. An array indexed by procedure identifier containing metadata for procedures and functions.
- Type descriptors. An array indexed by type identifier containing descriptions for types, both built-in and user defined types.
- Variable descriptors. An array indexed by variable identifier containing metadata for variables (e.g., their type identifiers).
- Program counter entries. An array mapping program counters to source file names and line numbers.

A procedure descriptor contains this information:

```
type Pent struct {
     name   string // for procedure/function
     addr   uint   // for its code in text
     nargs  int    // # of arguments
     nvars  int    // # of variables
     retsz  int    // size for return type or 0
```

```
	argsz  int    // size for arguments in stack
	varsz  int    // size for local vars in stack
	fname  string
	lineno int
	args   []Vent // Var descriptors for args
	vars   []Vent // Var descriptors for local vars.
}
```

A type descriptor contains enough to perform range checks, learn how to read values for the type, or write values for the type, learn the size for objects, and handle or dump objects for debugging.

```
type Tent struct {
	name   string   // of the type
	fmt    rune     // value format character
	first  int      // legal value or index
	last   int      // idem

	nitems int      // # of values or elements
	sz     uint     // in memory for values
	etid   uint     // element type id
	lits   []string // names for literals
	fields []Vent   // only name, tid, and addr defined
}
```

A variable descriptor is used to describe variables, mostly for debugging and stack dumps.

```
type Vent struct {
	name   string // of variable or constant
	tid    uint   // type
	addr   uint32 // in memory (offset for args, l.vars.)

	fname  string
	lineno int
	val    string // initial value as a string, or "".
	fields []Vent // aggregate members
}
```

Program counter entries have this information. Some fields are used to report leaks after program completion.

```
struct Pc
{
	ulong pc;
	char *fname;
	ulong lineno;
	Pc*  next;     /* Pc with leaks; for leaks */
	uint n;   /* # of leaks in this Pc; for leaks */
};
```

5.2. Instruction set

An instruction has two fields: an instruction code and an instruction type. The former describes the instruction. The later describes if it handles integers, floats, or memory addresses (in those cases when the instruction can do several of them). This is the instruction set:

```
add     daddr  eqm    idx    lt     mul    not    sto
addr    data   eqr    ind    ltr    mulr   or     stom
and     datar  fld    jmp    lvar   ne     pow    sub
arg     div    ge     jmpf   minus  nea    ptr    subr
call    divr   ger    jmpt   minusr nem    push
cast    eq     gt     le     mod    ner    pushr
castr   eqa    gtr    ler    modr   nop    ret
```

PAM instructions are described by this enumeration (explained later).

```
const (
     // instruction code (ic)
     ICnop   = iota // nop
     ICle           // le|r −sp −sp +sp
     ICge           // ge|r −sp −sp +sp
     ICpow          // pow|r −sp −sp +sp
     IClt           // lt|r −sp −sp +sp

     ICgt           // gt|r −sp −sp +sp
     ICmul          // mul|r −sp −sp +sp
     ICdiv          // div|r −sp −sp +spPBacos *.y
     ICmod          // mod|r −sp −sp +sp
     ICadd          // add|r −sp −sp +sp

     ICsub          // sub|r −sp −sp +sp
     ICminus        // minus|r −sp +sp
     ICnot          // not −sp +sp
     ICor           // or −sp −sp +sp
     ICand          // and −sp −sp +sp

     ICeq           // eq|r|a −sp −sp +sp
     ICne           // ne|r|a −sp −sp +sp
     ICptr          // ptr −sp +sp
     // obtain address for ptr in stack

     ICargs          // those after have an argument
     ICpush = ICargs // push|r n +sp
     // push n in the stack
)
const (
     ICindir = iota + ICpush + 1 // indir|a  n −sp +sp
     // replace address with referenced bytes
     ICjmp  // jmp addr
     ICjmpt // jmpt addr
     ICjmpf // jmpf addr

     ICidx  // idx tid  −sp −sp +sp
     // replace address[index] with elem. addr.
     ICfld // fld n −sp +sp
     // replace obj addr with field (at n) addr.
     ICdaddr // daddr n +sp
     // push address for data at n
     ICdata // data n +sp
```

```
	// push n bytes of data following instruction
	ICeqm // eqm n −sp −sp +sp
	// compare data pointed to by addresses
	ICnem // nem n −sp −sp +sp
	// compare data pointed to by addresses
	ICcall // call pid
	ICret  // ret pid
	ICarg  // arg n +sp
	// push address for arg object at n
	IClvar // lvar n +sp

	// push address for lvar object at n
	ICstom // stom tid −sp −sp
	// cp tid's sz bytes from address to address
	ICsto // sto tid −sp −sp
	// cp tid's sz bytes to address from stack
	ICcast // cast|r tid −sp +sp
	// convert int (or real |r) to type tid
)

	/* instr. type (it) */
	ITint  = 0
	ITaddr = 0x40
	ITreal = 0x80
	ITmask = ITreal | ITaddr
```

All instructions above *ICargs* (which is not an instruction) do not have a following argument in the program text. A single word contains the entire instruction. Those below use a following word to contain the argument for the instruction.

Instructions that have a suffix ``|r'' in their comment have a variant that knows how to handle reals. For example, the entry for *ICpush* means that there are two instructions: `push` and `pushr`. The former pushes an integer value (the argument) in the stack. The later pushes a float value in the stack.

Instructions with the suffix ``|a'' have a variant that handles addresses.

All atomic values in the stack (booleans, characters, integers, and floats) occupy a single word (32 bits). Addresses use 64 bits, to simplify execution in 64 bit environments. That is, addresses may be actual pointers. For example, there are three *eq* instructions: `eq`, `eqr`, and `eqa`: They compare integers, floats, and addresses (respectively).

Besides the argument in the program text, most instructions operate with stack arguments (and pop them off the stack) and push results back into the stack. This is represented by the ``+sp'' (push) and ``−sp'' in the description. Each one of the latter refers to a single argument taken from the stack.

5.3. Builtins

Builtin procedures and functions have addresses that are not procedure ids. Instead, they have the *PAMbuiltin* bit set and contain a builtin number in remaining bits:

```
// Builtin addresses
PAMbuiltin = 0x80000000,
```

```
const (
     PBacos = iota
     PBasin
     PBatan
     PBclose
     PBcos
     PBdispose // 0x5

     PBexp
     PBfatal
     PBfeof
     PBfeol
     PBfpeek // 0xa

     PBfread
     PBfreadeol
     PBfreadln
     PBfrewind
     PBfwrite // 0xf

     PBfwriteln
     PBfwriteeol
     PBlog
     PBlog10
     PBnew // 0x14

     PBopen
     PBpow
     PBpred
     PBsin
     PBsqrt // 0x19

     PBdata
     PBfflush
     PBgclear
     PBgclose
     PBgshowcursor
     PBgellipse

     PBgfillcol
     PBgfillrgb
     PBgkeypress
     PBgline
     PBgloc

     PBgopen
     PBgpencol
     PBgpenrgb
     PBgpenwidth
     PBgplay

     PBgpolygon
     PBgreadmouse
     PBgstop
     PBgtextheight
```

```
     PBrand
     PBsleep
     PBstack
     PBsucc
     PBtan
     Nbuiltins
)
```

The arguments for each builtin do not always match those supplied by the user. For example, file I/O procedures carry a type id besides the object or value to let PAM know how to read and write the argument (i.e., which is is its type descriptor). This is not documented here. See the implementation for the builtins in *pilib.c*.

5.4. Binary files.

A PAM binary is indeed a PAM assembly file and not a binary. It is a text file, both for debugging and for portability and pedagogical purposes.

The file must start with

```
#!/bin/pi
```

Lines starting with ‘‘#’’ are ignored. The second line must report the procedure id for *main*:

```
entry 3
```

for example. Following this, there are different sections for types, variables (and constants), procedures, text, and PC/source entries. Each section starts with a line that has the keyword *types*, *vars*, *procs*, *text*, and *pcs* (respectively) followed by the number of entries in the section. Each entry is a descriptor (see above) or a text instruction (perhaps with an argument in the same line).

Descriptors have the information shown in the structures found before in this document. Instructions have their address, instruction code (mnemonic, actually) and argument if any.

The compiler adds comments in the assembly file to match PAM instructions with the source code.

6. Example source

```
/*
 * Example program. Write the longest word in the input.
 */
program Word;

consts:
     Blocknc = 2;

types:
     Tblock = array[1..Blocknc] of char;
     Tword = ^Tnode;
     Tnode = record{
          block: Tblock;
          nc: int;
          next: Tword;
     };
```

```
function isblank(c: char): bool
{
     return c == ' ' or c == Tab or c == Eol;
}

procedure skipblanks(ref end: bool)
     c: char;
{
     do{
          peek(c);
          if(c == ' ' or c == '    '){
               read(c);
          }else if(c == Eol){
               readeol();
          }
     }while(not eof() and isblank(c));
     end = eof();
}

procedure initword(ref w: Tword)
{
     w = nil;
}

function wordnc(w: Tword): int
     tot: int;
{
     tot = 0;
     while(w != nil){
          tot = tot + w^.nc;
          w = w^.next;
     }
     return tot;
}

procedure writeword(w: Tword)
     i: int;
{
     write("'");
     while(w != nil){
          for(i = 1, i <= w^.nc){
               write(w^.block[i]);
          }
          w = w^.next;
     }
     write("'");
}

procedure mkblock(ref w: Tword)
{
     new(w);
     w^.nc = 0;
     w^.next = nil;
}
```

```
procedure addtoword(ref w: Tword, c: char)
     p: Tword;
{
     if(w == nil){
          mkblock(w);
     }
     p = w;
     while(p^.next != nil){
          p = p^.next;
     }
     if(p^.nc == Blocknc){
          mkblock(p^.next);
          p = p^.next;
     }
     p^.nc = p^.nc + 1;
     p^.block[p^.nc] = c;
}

procedure delword(ref w: Tword)
{
     if(w != nil){
          delword(w^.next);
          dispose(w);
          initword(w);
     }
}

procedure readword(ref w: Tword)
     c: char;
{
     do{
          read(c);
          addtoword(w, c);
          peek(c);
     }while(not eof() and not isblank(c));

}

function wordchar(w: Tword, n: int): char
     c: char;
{
     c = '?';
     while(n > 0 and w != nil){
          if(n <= Blocknc){
               c = w^.block[n];
               n = 0;
          }else{
               n = n - Blocknc;
               w = w^.next;
          }
     }
     return c;
}
```

```
procedure cpword(ref dw: Tword, sw: Tword)
     i: int;
{
     delword(dw);
     for(i = 1, i <= wordnc(sw)){
          addtoword(dw, wordchar(sw, i));
     }
}

procedure main()
     done: bool;
     w: Tword;
     max: Tword;
{
     initword(max);
     do{
          skipblanks(done);
          if(not done){
               initword(w);
               readword(w);
               if(wordnc(w) > wordnc(max)){
                    cpword(max, w);
               }
               delword(w);
          }
     }while(not eof());
     writeword(max);
     write(" with len ");
     writeln(wordnc(max));
     delword(max);
}
```

7. Example binary

This is the binary file produced for the source in the previous section.

```
#!/bin/pam
entry 11
types 17
0 bool b  0 1 2 4 0
1 char c  0 255 256 4 0
2 int i  -2147483646 2147483647 0 4 0
3 float r  0 0 0 4 0
4 $nil p  0 0 0 8 0
5 file f  0 0 0 4 0
6 strength h  0 255 0 4 0
7 opacity l  0 1 0 4 0
8 color e  0 6 7 4 0
Black
Red
Green
Blue
Yellow
Orange
White
9 button u  0 255 0 4 0
10 sound e  0 19 20 4 0
Woosh
Beep
Sheep
Phaser
Rocket
CNote
CsharpNote
DNote
DsharpNote
ENote
FNote
FsharpNote
GNote
GsharpNote
ANote
AsharpNote
BNote
Bomb
Fail
Tada
11 $range1 i  1 2 2 4 0
12 Tblock a  1 2 2 8 1
13 Tword p  0 0 0 8 14
14 Tnode R  0 0 3 20 0
block 12 0x0
nc 2 0x8
next 13 0xc
15 $tstr1 s  0 0 1 4 1
16 $tstr10 s  0 9 10 40 1
vars 31
Maxint 2 0x0 2147483647 'example.p' 4
Minint 2 0x4 -2147483646 'example.p' 4
Maxchar 1 0x8 255 'example.p' 4
Minchar 1 0xc 0 'example.p' 4
Minstrength 6 0x10 0 'example.p' 4
Maxstrength 6 0x14 255 'example.p' 4
Transp 7 0x18 0 'example.p' 4
Tlucid 7 0x1c 0 'example.p' 4
Opaque 7 0x20 0 'example.p' 4
```

```
NoBut 9 0x24 0 ’example.p’ 4
Esc 1 0x28 27 ’example.p’ 4
Shift 1 0x2c 241 ’example.p’ 4
Return 1 0x30 246 ’example.p’ 4
Tab 1 0x34 9 ’example.p’ 21
Up 1 0x38 245 ’example.p’ 4
Right 1 0x3c 242 ’example.p’ 4
Ctrl 1 0x40 240 ’example.p’ 4
MetaRight 1 0x44 248 ’example.p’ 4
MetaLeft 1 0x48 247 ’example.p’ 4
Eof 1 0x4c 255 ’example.p’ 4
Down 1 0x50 244 ’example.p’ 4
Del 1 0x54 249 ’example.p’ 4
Eol 1 0x58 10 ’example.p’ 31
Nul 1 0x5c 0 ’example.p’ 4
Left 1 0x60 243 ’example.p’ 4
Blocknc 2 0x64 2 ’example.p’ 121
$s0 15 0x68 ’’’’ ’example.p’ 57
$s1 15 0x6c ’’’’ ’example.p’ 64
$s2 16 0x70 ’ with len ’ ’example.p’ 155
stdin 5 0x98 – ’example.p’ 4
stdout 5 0x9c – ’example.p’ 4
procs 12
0 isblank 0x00000 1 0 4 4 0 ’example.p’ 108
c 1 0x0 – ’example.p’ 21
1 skipblanks 0x00019 1 1 0 8 4 ’example.p’ 144
end 0 0x0 – ’example.p’ 35
c 1 0x0 – ’example.p’ 34
2 initword 0x0006b 1 0 0 8 0 ’example.p’ 146
w 13 0x0 – ’example.p’ 40
3 wordnc 0x00077 1 1 4 8 4 ’example.p’ 156
w 13 0x0 – ’example.p’ 49
tot 2 0x0 – ’example.p’ 51
4 writeword 0x000ad 1 1 0 8 4 ’example.p’ 154
w 13 0x0 – ’example.p’ 62
i 2 0x0 – ’example.p’ 60
5 mkblock 0x00126 1 0 0 8 0 ’example.p’ 85
w 13 0x0 – ’example.p’ 71
6 addtoword 0x0014c 2 1 0 12 8 ’example.p’ 133
w 13 0x4 – ’example.p’ 80
c 1 0x0 – ’example.p’ 89
p 13 0x0 – ’example.p’ 89
7 delword 0x001d1 1 0 0 8 0 ’example.p’ 157
w 13 0x0 – ’example.p’ 97
8 readword 0x001f7 1 1 0 8 4 ’example.p’ 147
w 13 0x0 – ’example.p’ 106
c 1 0x0 – ’example.p’ 108
9 wordchar 0x00226 2 1 4 12 4 ’example.p’ 133
w 13 0x4 – ’example.p’ 122
n 2 0x0 – ’example.p’ 121
c 1 0x0 – ’example.p’ 125
10 cpword 0x0027d 2 1 0 16 4 ’example.p’ 149
dw 13 0x8 – ’example.p’ 133
sw 13 0x0 – ’example.p’ 133
i 2 0x0 – ’example.p’ 133
11 main 0x002c1 0 3 0 0 20 ’example.p’ 137
done 0 0x0 – ’example.p’ 145
w 13 0x4 – ’example.p’ 151
max 13 0xc – ’example.p’ 157
text 802
```

```
# isblank()
# {...}
# return or(or(==($c: char, ' '), ==($c: char, Tab=Tab)), ==($c: char, Eol=Eol))
00000     push 0x000000000a   # Eol=Eol;
00002     arg  0x0000000000   # $c: char;
00004     ind  0x0000000004
00006     eq
00007     push 0x0000000009   # Tab=Tab;
00009     arg  0x0000000000   # $c: char;
0000b     ind  0x0000000004
0000d     eq
0000e     push 0x0000000020   # ' ';
00010     arg  0x0000000000   # $c: char;
00012     ind  0x0000000004
00014     eq
00015     or
00016     or
00017     ret  0x0000000000
# skipblanks()
# {...}
# dowhile(and(not(feof(stdin: file)), isblank(%c: char)))
# {...}
# fpeek(stdin: file, %c: char)
00019     lvar 0x0000000000   # %c: char;
0001b     daddr     0x0000000098   # stdin: file;
0001d     ind  0x0000000004
0001f     call 0x008000000a   # fpeek();
# if(or(==(%c: char, ' '), ==(%c: char, Tab)))
00021     push 0x0000000009   # Tab;
00023     lvar 0x0000000000   # %c: char;
00025     ind  0x0000000004
00027     eq
00028     push 0x0000000020   # ' ';
0002a     lvar 0x0000000000   # %c: char;
0002c     ind  0x0000000004
0002e     eq
0002f     or
00030     jmpf 0x000000003e
# {...}
# fread(stdin: file, %c: char)
00032     lvar 0x0000000000   # %c: char;
00034     daddr     0x0000000098   # stdin: file;
00036     ind  0x0000000004
00038     push 0x0000000001
0003a     call 0x008000000b   # fread();
0003c     jmp  0x000000004d
# if(==(%c: char, Eol=Eol))
0003e     push 0x000000000a   # Eol=Eol;
00040     lvar 0x0000000000   # %c: char;
00042     ind  0x0000000004
00044     eq
00045     jmpf 0x000000004d
# {...}
# freadeol(stdin: file)
00047     daddr     0x0000000098   # stdin: file;
00049     ind  0x0000000004
0004b     call 0x008000000c   # freadeol();
0004d     lvar 0x0000000000   # %c: char;
0004f     ind  0x0000000004
00051     call 0x0000000000   # isblank();
```

```
00053     daddr     0x0000000098   # stdin: file;
00055     ind  0x0000000004
00057     call 0x0080000008   # feof();
00059     not
0005a     and
0005b     jmpt 0x0000000019
# &end: bool = feof(stdin: file)
0005d     daddr     0x0000000098   # stdin: file;
0005f     ind  0x0000000004
00061     call 0x0080000008   # feof();
00063     arg  0x0000000000   # &end: bool;
00065     ind  0x0000000008
00067     sto  0x0000000000
# return <nil>
00069     ret  0x0000000001
# initword()
# {...}
# &w: Tword = nil
0006b     data 0x0000000008   # nil;
0006d     0x0
0006e     0x0
0006f     arg  0x0000000000   # &w: Tword;
00071     ind  0x0000000008
00073     sto  0x000000000d
# return <nil>
00075     ret  0x0000000002
# wordnc()
# {...}
# %tot: int = 0
00077     push 0x0000000000   # 0;
00079     lvar 0x0000000000   # %tot: int;
0007b     sto  0x0000000002
# while(!=($w: Tword, nil))
0007d     data 0x0000000008   # nil;
0007f     0x0
00080     0x0
00081     arg  0x0000000000   # $w: Tword;
00083     ind  0x0000000008
00085     nea
00086     jmpf 0x00000000a7
# {...}
# %tot: int = +(%tot: int, .(^($w: Tword), nc: int))
00088     arg  0x0000000000   # .; ^; $w: Tword;
0008a     ind  0x0000000008
0008c     ptr
0008d     fld  0x0000000008
0008f     ind  0x0000000004
00091     lvar 0x0000000000   # %tot: int;
00093     ind  0x0000000004
00095     add
00096     lvar 0x0000000000   # %tot: int;
00098     sto  0x0000000002
# $w: Tword = .(^($w: Tword), next: Tword)
0009a     arg  0x0000000000   # .; ^; $w: Tword;
0009c     ind  0x0000000008
0009e     ptr
0009f     fld  0x000000000c
000a1     arg  0x0000000000   # $w: Tword;
000a3     stom 0x000000000d
000a5     jmp  0x000000007d
```

```
# return %tot: int
000a7     lvar 0x0000000000   # %tot: int;
000a9     ind  0x0000000004
000ab     ret  0x0000000003
# writeword()
# {...}
# fwrite(stdout: file, $s0="'")
000ad     daddr     0x0000000068   # $s0="'";
000af     ind  0x0000000004
000b1     daddr     0x000000009c   # stdout: file;
000b3     ind  0x0000000004
000b5     push 0x000000000f
000b7     call 0x008000000f   # fwrite();
# while(!=($w: Tword, nil))
000b9     data 0x0000000008   # nil;
000bb     0x0
000bc     0x0
000bd     arg  0x0000000000   # $w: Tword;
000bf     ind  0x0000000008
000c1     nea
000c2     jmpf 0x0000000118
# {...}
# {...}
# %i: int = 1
000c4     push 0x0000000001   # 1;
000c6     lvar 0x0000000000   # %i: int;
000c8     sto  0x0000000002
# for(<=(%i: int, .(^($w: Tword), nc: int)))
000ca     arg  0x0000000000   # .; ^; $w: Tword;
000cc     ind  0x0000000008
000ce     ptr
000cf     fld  0x0000000008
000d1     ind  0x0000000004
000d3     lvar 0x0000000000   # %i: int;
000d5     ind  0x0000000004
000d7     le
000d8     jmpf 0x000000010b
# {...}
# fwrite(stdout: file, [](.(^($w: Tword), block: Tblock), %i: int))
000da     lvar 0x0000000000   # []; %i: int;
000dc     ind  0x0000000004
000de     arg  0x0000000000   # .; ^; $w: Tword;
000e0     ind  0x0000000008
000e2     ptr
000e3     idx  0x000000000c
000e5     ind  0x0000000004
000e7     daddr     0x000000009c   # stdout: file;
000e9     ind  0x0000000004
000eb     push 0x0000000001
000ed     call 0x008000000f   # fwrite();
000ef     arg  0x0000000000   # .; ^; $w: Tword;
000f1     ind  0x0000000008
000f3     ptr
000f4     fld  0x0000000008
000f6     ind  0x0000000004
000f8     lvar 0x0000000000   # %i: int;
000fa     ind  0x0000000004
000fc     eq
000fd     jmpt 0x000000010b
# %i: int = succ(%i: int)
```

```
000ff     lvar 0x0000000000   # %i: int;
00101     ind  0x0000000004
00103     call 0x0080000031   # succ();
00105     lvar 0x0000000000   # %i: int;
00107     sto  0x0000000002
00109     jmp  0x00000000ca
# $w: Tword = .(^($w: Tword), next: Tword)
0010b     arg  0x0000000000   # .; ^; $w: Tword;
0010d     ind  0x0000000008
0010f     ptr
00110     fld  0x000000000c
00112     arg  0x0000000000   # $w: Tword;
00114     stom 0x000000000d
00116     jmp  0x00000000b9
# fwrite(stdout: file, $s1="'")
00118     daddr     0x000000006c   # $s1="'";
0011a     ind  0x0000000004
0011c     daddr     0x000000009c   # stdout: file;
0011e     ind  0x0000000004
00120     push 0x000000000f
00122     call 0x008000000f   # fwrite();
# return <nil>
00124     ret  0x0000000004
# mkblock()
# {...}
# new(&w: Tword)
00126     arg  0x0000000000   # &w: Tword;
00128     ind  0x0000000008
0012a     push 0x000000000d
0012c     call 0x0080000014   # new();
# .(^(&w: Tword), nc: int) = 0
0012e     push 0x0000000000   # 0;
00130     arg  0x0000000000   # .; ^; &w: Tword;
00132     ind  0x0000000008
00134     ind  0x0000000008
00136     ptr
00137     fld  0x0000000008
00139     sto  0x0000000002
# .(^(&w: Tword), next: Tword) = nil
0013b     data 0x0000000008   # nil;
0013d     0x0
0013e     0x0
0013f     arg  0x0000000000   # .; ^; &w: Tword;
00141     ind  0x0000000008
00143     ind  0x0000000008
00145     ptr
00146     fld  0x000000000c
00148     sto  0x000000000d
# return <nil>
0014a     ret  0x0000000005
# addtoword()
# {...}
# if(==(&w: Tword, nil))
0014c     data 0x0000000008   # nil;
0014e     0x0
0014f     0x0
00150     arg  0x0000000004   # &w: Tword;
00152     ind  0x0000000008
00154     ind  0x0000000008
00156     eqa
```

```
00157     jmpf 0x000000015f
# {...}
# mkblock(&w: Tword)
00159     arg  0x0000000004   # &w: Tword;
0015b     ind  0x0000000008
0015d     call 0x0000000005   # mkblock();
# %p: Tword = &w: Tword
0015f     arg  0x0000000004   # &w: Tword;
00161     ind  0x0000000008
00163     lvar 0x0000000000   # %p: Tword;
00165     stom 0x000000000d
# while(!=(.(^(%p: Tword), next: Tword), nil))
00167     data 0x0000000008   # nil;
00169     0x0
0016a     0x0
0016b     lvar 0x0000000000   # .; ^; %p: Tword;
0016d     ind  0x0000000008
0016f     ptr
00170     fld  0x000000000c
00172     ind  0x0000000008
00174     nea
00175     jmpf 0x0000000184
# {...}
# %p: Tword = .(^(%p: Tword), next: Tword)
00177     lvar 0x0000000000   # .; ^; %p: Tword;
00179     ind  0x0000000008
0017b     ptr
0017c     fld  0x000000000c
0017e     lvar 0x0000000000   # %p: Tword;
00180     stom 0x000000000d
00182     jmp  0x0000000167
# if(==(.(^(%p: Tword), nc: int), Blocknc=2))
00184     push 0x0000000002   # Blocknc=2;
00186     lvar 0x0000000000   # .; ^; %p: Tword;
00188     ind  0x0000000008
0018a     ptr
0018b     fld  0x0000000008
0018d     ind  0x0000000004
0018f     eq
00190     jmpf 0x00000001a6
# {...}
# mkblock(.(^(%p: Tword), next: Tword))
00192     lvar 0x0000000000   # .; ^; %p: Tword;
00194     ind  0x0000000008
00196     ptr
00197     fld  0x000000000c
00199     call 0x0000000005   # mkblock();
# %p: Tword = .(^(%p: Tword), next: Tword)
0019b     lvar 0x0000000000   # .; ^; %p: Tword;
0019d     ind  0x0000000008
0019f     ptr
001a0     fld  0x000000000c
001a2     lvar 0x0000000000   # %p: Tword;
001a4     stom 0x000000000d
# .(^(%p: Tword), nc: int) = +(.(^(%p: Tword), nc: int), 1)
001a6     push 0x0000000001   # 1;
001a8     lvar 0x0000000000   # .; ^; %p: Tword;
001aa     ind  0x0000000008
001ac     ptr
001ad     fld  0x0000000008
```

```
001af     ind  0x0000000004
001b1     add
001b2     lvar 0x0000000000   # .; ^; %p: Tword;
001b4     ind  0x0000000008
001b6     ptr
001b7     fld  0x0000000008
001b9     sto  0x0000000002
# [](.(^(%p: Tword), block: Tblock), .(^(%p: Tword), nc: int)) = $c: char
001bb     arg  0x0000000000   # $c: char;
001bd     lvar 0x0000000000   # []; .; ^; %p: Tword;
001bf     ind  0x0000000008
001c1     ptr
001c2     fld  0x0000000008
001c4     ind  0x0000000004
001c6     lvar 0x0000000000   # .; ^; %p: Tword;
001c8     ind  0x0000000008
001ca     ptr
001cb     idx  0x000000000c
001cd     stom 0x0000000001
# return <nil>
001cf     ret  0x0000000006
# delword()
# {...}
# if(!=(&w: Tword, nil))
001d1     data 0x0000000008   # nil;
001d3     0x0
001d4     0x0
001d5     arg  0x0000000000   # &w: Tword;
001d7     ind  0x0000000008
001d9     ind  0x0000000008
001db     nea
001dc     jmpf 0x00000001f5
# {...}
# delword(.(^(&w: Tword), next: Tword))
001de     arg  0x0000000000   # .; ^; &w: Tword;
001e0     ind  0x0000000008
001e2     ind  0x0000000008
001e4     ptr
001e5     fld  0x000000000c
001e7     call 0x0000000007   # delword();
# dispose(&w: Tword)
001e9     arg  0x0000000000   # &w: Tword;
001eb     ind  0x0000000008
001ed     call 0x0080000005   # dispose();
# initword(&w: Tword)
001ef     arg  0x0000000000   # &w: Tword;
001f1     ind  0x0000000008
001f3     call 0x0000000002   # initword();
# return <nil>
001f5     ret  0x0000000007
# readword()
# {...}
# dowhile(and(not(feof(stdin: file)), not(isblank(%c: char))))
# {...}
# fread(stdin: file, %c: char)
001f7     lvar 0x0000000000   # %c: char;
001f9     daddr     0x0000000098   # stdin: file;
001fb     ind  0x0000000004
001fd     push 0x0000000001
001ff     call 0x008000000b   # fread();
```

```
# addtoword(&w: Tword, %c: char)
00201     lvar 0x0000000000   # %c: char;
00203     ind  0x0000000004
00205     arg  0x0000000000   # &w: Tword;
00207     ind  0x0000000008
00209     call 0x0000000006   # addtoword();
# fpeek(stdin: file, %c: char)
0020b     lvar 0x0000000000   # %c: char;
0020d     daddr     0x0000000098   # stdin: file;
0020f     ind  0x0000000004
00211     call 0x008000000a   # fpeek();
00213     lvar 0x0000000000   # %c: char;
00215     ind  0x0000000004
00217     call 0x0000000000   # isblank();
00219     not
0021a     daddr     0x0000000098   # stdin: file;
0021c     ind  0x0000000004
0021e     call 0x0080000008   # feof();
00220     not
00221     and
00222     jmpt 0x00000001f7
# return <nil>
00224     ret  0x0000000008
# wordchar()
# {...}
# %c: char = '?'
00226     push 0x000000003f   # '?';
00228     lvar 0x0000000000   # %c: char;
0022a     sto  0x0000000001
# while(and(>($n: int, 0), !=($w: Tword, nil)))
0022c     data 0x0000000008   # nil;
0022e     0x0
0022f     0x0
00230     arg  0x0000000004   # $w: Tword;
00232     ind  0x0000000008
00234     nea
00235     push 0x0000000000   # 0;
00237     arg  0x0000000000   # $n: int;
00239     ind  0x0000000004
0023b     gt
0023c     and
0023d     jmpf 0x0000000277
# {...}
# if(<=($n: int, Blocknc=2))
0023f     push 0x0000000002   # Blocknc=2;
00241     arg  0x0000000000   # $n: int;
00243     ind  0x0000000004
00245     le
00246     jmpf 0x000000025f
# {...}
# %c: char = [](.(^($w: Tword), block: Tblock), $n: int)
00248     arg  0x0000000000   # []; $n: int;
0024a     ind  0x0000000004
0024c     arg  0x0000000004   # .; ^; $w: Tword;
0024e     ind  0x0000000008
00250     ptr
00251     idx  0x000000000c
00253     lvar 0x0000000000   # %c: char;
00255     stom 0x0000000001
# $n: int = 0
```

```
00257     push 0x0000000000   # 0;
00259     arg  0x0000000000   # $n: int;
0025b     sto  0x0000000002
0025d     jmp  0x0000000275
# else
# $n: int = −($n: int, Blocknc=2)
0025f     push 0x0000000002   # Blocknc=2;
00261     arg  0x0000000000   # $n: int;
00263     ind  0x0000000004
00265     sub
00266     arg  0x0000000000   # $n: int;
00268     sto  0x0000000002
# $w: Tword = .(^($w: Tword), next: Tword)
0026a     arg  0x0000000004   # .; ^; $w: Tword;
0026c     ind  0x0000000008
0026e     ptr
0026f     fld  0x000000000c
00271     arg  0x0000000004   # $w: Tword;
00273     stom 0x000000000d
00275     jmp  0x000000022c
# return %c: char
00277     lvar 0x0000000000   # %c: char;
00279     ind  0x0000000004
0027b     ret  0x0000000009
# cpword()
# {...}
# delword(&dw: Tword)
0027d     arg  0x0000000008   # &dw: Tword;
0027f     ind  0x0000000008
00281     call 0x0000000007   # delword();
# {...}
# %i: int = 1
00283     push 0x0000000001   # 1;
00285     lvar 0x0000000000   # %i: int;
00287     sto  0x0000000002
# for(<=(%i: int, wordnc($sw: Tword)))
00289     arg  0x0000000000   # $sw: Tword;
0028b     ind  0x0000000008
0028d     call 0x0000000003   # wordnc();
0028f     lvar 0x0000000000   # %i: int;
00291     ind  0x0000000004
00293     le
00294     jmpf 0x00000002bf
# {...}
# addtoword(&dw: Tword, wordchar($sw: Tword, %i: int))
00296     lvar 0x0000000000   # %i: int;
00298     ind  0x0000000004
0029a     arg  0x0000000000   # $sw: Tword;
0029c     ind  0x0000000008
0029e     call 0x0000000009   # wordchar();
002a0     arg  0x0000000008   # &dw: Tword;
002a2     ind  0x0000000008
002a4     call 0x0000000006   # addtoword();
002a6     arg  0x0000000000   # $sw: Tword;
002a8     ind  0x0000000008
002aa     call 0x0000000003   # wordnc();
002ac     lvar 0x0000000000   # %i: int;
002ae     ind  0x0000000004
002b0     eq
002b1     jmpt 0x00000002bf
```

```
# %i: int = succ(%i: int)
002b3     lvar 0x0000000000   # %i: int;
002b5     ind  0x0000000004
002b7     call 0x0080000031   # succ();
002b9     lvar 0x0000000000   # %i: int;
002bb     sto  0x0000000002
002bd     jmp  0x0000000289
# return <nil>
002bf     ret  0x000000000a
# main()
# {...}
# initword(%max: Tword)
002c1     lvar 0x000000000c   # %max: Tword;
002c3     call 0x0000000002   # initword();
# dowhile(not(feof(stdin: file)))
# {...}
# skipblanks(%done: bool)
002c5     lvar 0x0000000000   # %done: bool;
002c7     call 0x0000000001   # skipblanks();
# if(not(%done: bool))
002c9     lvar 0x0000000000   # %done: bool;
002cb     ind  0x0000000004
002cd     not
002ce     jmpf 0x00000002f3
# {...}
# initword(%w: Tword)
002d0     lvar 0x0000000004   # %w: Tword;
002d2     call 0x0000000002   # initword();
# readword(%w: Tword)
002d4     lvar 0x0000000004   # %w: Tword;
002d6     call 0x0000000008   # readword();
# if(>(wordnc(%w: Tword), wordnc(%max: Tword)))
002d8     lvar 0x000000000c   # %max: Tword;
002da     ind  0x0000000008
002dc     call 0x0000000003   # wordnc();
002de     lvar 0x0000000004   # %w: Tword;
002e0     ind  0x0000000008
002e2     call 0x0000000003   # wordnc();
002e4     gt
002e5     jmpf 0x00000002ef
# {...}
# cpword(%max: Tword, %w: Tword)
002e7     lvar 0x0000000004   # %w: Tword;
002e9     ind  0x0000000008
002eb     lvar 0x000000000c   # %max: Tword;
002ed     call 0x000000000a   # cpword();
# delword(%w: Tword)
002ef     lvar 0x0000000004   # %w: Tword;
002f1     call 0x0000000007   # delword();
002f3     daddr     0x0000000098   # stdin: file;
002f5     ind  0x0000000004
002f7     call 0x0080000008   # feof();
002f9     not
002fa     jmpt 0x00000002c5
# writeword(%max: Tword)
002fc     lvar 0x000000000c   # %max: Tword;
002fe     ind  0x0000000008
00300     call 0x0000000004   # writeword();
# fwrite(stdout: file, $s2=" with len ")
00302     daddr     0x0000000070   # $s2=" with len ";
```

```
00304     ind  0x0000000028
00306     daddr     0x000000009c   # stdout: file;
00308     ind  0x0000000004
0030a     push 0x0000000010
0030c     call 0x008000000f   # fwrite();
# fwriteln(stdout: file, wordnc(%max: Tword))
0030e     lvar 0x000000000c   # %max: Tword;
00310     ind  0x0000000008
00312     call 0x0000000003   # wordnc();
00314     daddr     0x000000009c   # stdout: file;
00316     ind  0x0000000004
00318     push 0x0000000002
0031a     call 0x0080000010   # fwriteln();
# delword(%max: Tword)
0031c     lvar 0x000000000c   # %max: Tword;
0031e     call 0x0000000007   # delword();
# return <nil>
00320     ret  0x000000000b
pcs 75
00000     'example.p'    21
00019     'example.p'    28
00021     'example.p'    29
00032     'example.p'    30
0003e     'example.p'    31
00047     'example.p'    32
0005d     'example.p'    35
00069     'example.p'    159
0006b     'example.p'    40
00075     'example.p'    159
00077     'example.p'    46
0007d     'example.p'    47
00088     'example.p'    48
0009a     'example.p'    49
000a7     'example.p'    51
000ad     'example.p'    57
000b9     'example.p'    58
000c4     'example.p'    60
000c4     'example.p'    61
000da     'example.p'    60
000ff     'example.p'    61
0010b     'example.p'    62
00118     'example.p'    64
00124     'example.p'    159
00126     'example.p'    69
0012e     'example.p'    70
0013b     'example.p'    71
0014a     'example.p'    159
0014c     'example.p'    77
00159     'example.p'    78
0015f     'example.p'    80
00167     'example.p'    81
00177     'example.p'    82
00184     'example.p'    84
00192     'example.p'    85
0019b     'example.p'    86
001a6     'example.p'    88
001bb     'example.p'    89
001cf     'example.p'    159
001d1     'example.p'    94
001de     'example.p'    95
```

```
001e9     'example.p'    96
001ef     'example.p'    97
001f5     'example.p'    159
001f7     'example.p'    105
00201     'example.p'    106
0020b     'example.p'    107
00224     'example.p'    159
00226     'example.p'    115
0022c     'example.p'    116
0023f     'example.p'    117
00248     'example.p'    118
00257     'example.p'    119
0025f     'example.p'    121
0026a     'example.p'    122
00277     'example.p'    125
0027d     'example.p'    131
00283     'example.p'    133
00283     'example.p'    134
00296     'example.p'    133
002b3     'example.p'    134
002bf     'example.p'    159
002c1     'example.p'    142
002c5     'example.p'    144
002c9     'example.p'    145
002d0     'example.p'    146
002d4     'example.p'    147
002d8     'example.p'    148
002e7     'example.p'    149
002ef     'example.p'    151
002fc     'example.p'    154
00302     'example.p'    155
0030e     'example.p'    156
0031c     'example.p'    157
00320     'example.p'    159
```

8. Example graphical program

```
program ball;

/*
 * Graphical example program. Clasical bouncing ball in a rectangle.
 */

types:
     TypeVect = record {
          x: int;
          y: int;
     };

     TypeBall = record {
          pos: TypeVect;
          speed: TypeVect;
     };
```

```
consts:
     TQuantum = 50; /* milliseconds */
     SpeedScale = 50; /* divisor for milliseconds */
     SizeX = 5000;
     SizeY = 5000;
     SpeedX = -20;
     SpeedY = 43;
     BallRad = 100;
     Ball = TypeBall(TypeVect(BallRad, BallRad), TypeVect(SpeedX, SpeedY));

function sumvect(v1: TypeVect, v2: TypeVect): TypeVect
     s: TypeVect;
{
     s.x = v1.x+v2.x;
     s.y = v1.y+v2.y;
     return s;
}

function scalevect(v: TypeVect, l: int): TypeVect
     s: TypeVect;
{
     s.x = v.x*l;
     s.y = v.y*l;
     return s;
}

procedure reflect(ref b: TypeBall)
{
     if(b.pos.x < 0){
          b.pos.x = 0;
          b.speed.x = -b.speed.x;
     }else     if(b.pos.x > SizeX){
          b.pos.x = SizeX;
          b.speed.x = -b.speed.x;
     }
     if(b.pos.y < 0){
          b.pos.y = 0;
          b.speed.y = -b.speed.y;
     }else     if(b.pos.y > SizeY){
          b.pos.y = SizeY;
          b.speed.y = -b.speed.y;
     }
}

procedure update(ref b: TypeBall)
{
     b.pos = sumvect(b.pos, scalevect(b.speed, TQuantum/SpeedScale));
     reflect(b);
}
```

```
procedure drawball(g: file, ref b: TypeBall)
     x: int;
     y: int;
{
     gfillcol(g, Green, Opaque);
     gpencol(g, Red, Opaque);
     gpenwidth(g, 1);
     x = b.pos.x;
     y = b.pos.y;

     gellipse(g, x, y, BallRad, BallRad, 0.0);
}

procedure main()
     b: TypeBall;
     g: file;
     k: char;
{
     b = Ball;
     gopen(g, "ball");
     do{
          update(b);
          gclear(g);
          drawball(g, b);
          fflush(g);
          gkeypress(g, k);
          sleep(TQuantum);
     }while(not feof(g) and k != 'q');
     gclose(g);
}
```

The user interface can be seen in Figure 1.

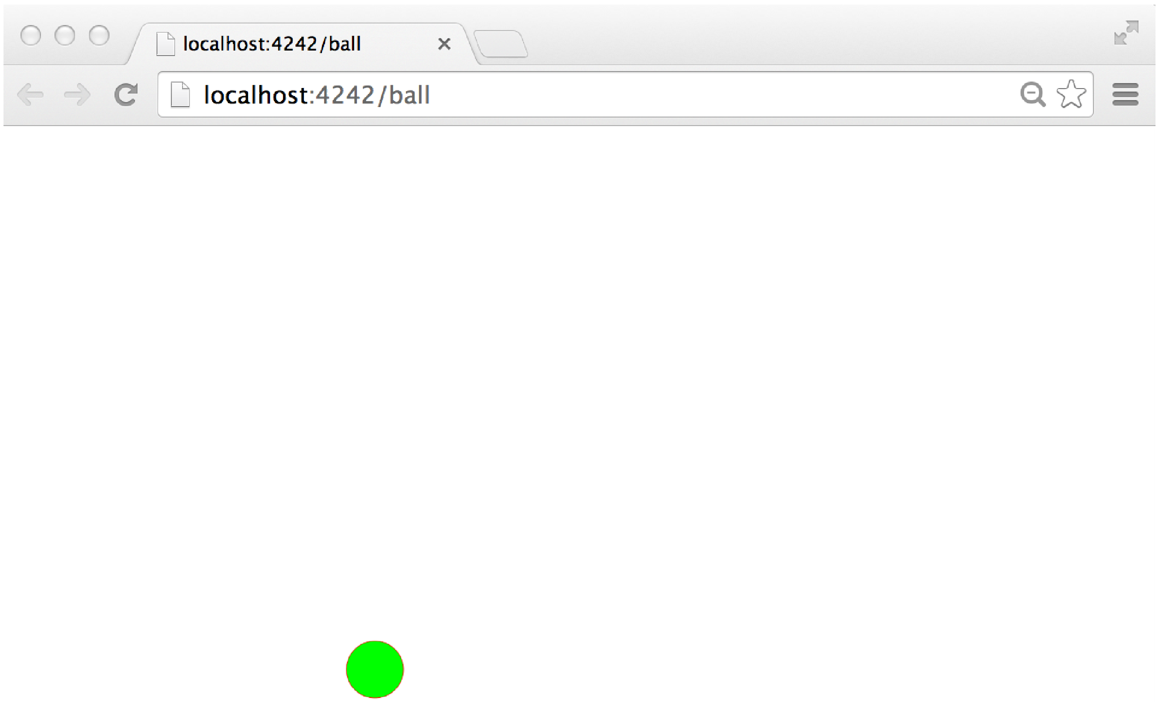

Figure 1: UI of the bouncing ball program