YAPL language specification

Overview

Yet another programming language (YAPL) Is a procedural 3GL in the style of C or BCPL. It features traditional blocks defined by curly braces and the usual 3GL control flow keywords and expressions. it differs from the C language in supporting only machine types like BCPL, although unlike BCPL and Bliss the machine types do include bytes and explicit pointers. YAPL does not have a preprocessor, featuring instead a stronger compile time expression evaluator so that the YAPL keyword **`const`** substitutes for #define symbols in C. **`Const`** values may be used for conditional compilation. For visibility, YAPL adopts the Golang convention: identifiers beginning with an uppercase letter are public. Other top-level identifiers are private to the compilation unit which is the source file.The lifetime of variables declared at the compilation unit level is the life of the program. All other variables are local to a block, stack-allocated, and exist only while their containing block is active.

This spec assumes the reader is familiar with the C and Go programming languages.

The initial compiler for YAPL was written by Claude Code in Golang. The compiler will be translated to YAPL and self-hosted in the future. The compiler's design owes a great deal to the book **A Retargetable C Compiler:Design and Implementation** by Fraser and Hanson and to their open source compiler project, lcc (https://github.com/drh/lcc) Claude: there is no need for you to read these–they were for my benefit.

Source code representation

YAPL Is a “retro” programming language. Source code is ASCII or, equivalently, UTF-8 code points 32 through 127. As a concession to modernity, quoted strings are encoded in UTF-8. Case is significant. Aside the white space characters defined below, ASCII control characters including NUL are not allowed in the source code except when they occur in quoted strings as the result of escape sequences.

Letters and digits¶

The underscore character _ (\x5F) is considered a lowercase letter.

```
letter        = ascii_letter | "_" .
decimal_digit = "0" … "9" .
binary_digit  = "0" | "1" .
octal_digit   = "0" … "7" .
hex_digit     = "0" … "9" | "A" … "F" | "a" … "f" .
```

Notation¶

T
he syntax is specified using a variant of Extended Backus-Naur Form (EBNF):

```
Syntax      = { Production } .
Production  = production_name "=" [ Expression ] "." .
Expression  = Term { "|" Term } .
Term        = Factor { Factor } .
Factor      = production_name | token [ "…" token ] | Group | Option |
Repetition .
Group       = "(" Expression ")" .
Option      = "[" Expression "]" .
Repetition  = "{" Expression "}" .
Productions are expressions constructed from terms and the following
operators, in increasing precedence:
```

`|`	alternation
`()`	grouping
`[]`	option (0 or 1 times)
`{}`	repetition (0 to n times)

Lowercase production names are used to identify lexical (terminal) tokens. Non-terminals are in CamelCase. Lexical tokens are enclosed in double quotes "" or back quotes ``. The form `a … b` represents the set of characters from `a` through `b` as alternatives. The horizontal ellipsis … is also used elsewhere in the spec to informally denote various enumerations or code snippets that are not further specified.

Characters

```
newline        = /* ASCII line feed*/ .
acii_char   = /* an ASCII char except newline */ .
```

The underscore character _ is considered a lowercase letter.

```
letter        = ascii_letter | "_" .
ascii_letter = "a" … "z" | "A" … "Z".
decimal_digit = "0" … "9" .
binary_digit  = "0" | "1" .
```

```
octal_digit    = "0" … "7" .
hex_digit      = "0" … "9" | "A" … "F" | "a" … "f" .
```

Lexical Elements

Tokens

There are four classes of tokens: *identifiers*, *keywords*, *operators and punctuation*, and *literals*.Tokens are separated by *white space or comments*. They are assembled from the source code by a “greedy” algorithm:. when breaking the input source into tokens, the next token is the longest sequence of characters that form a valid token.

White Space

ASCII space, tab, carriage return and line feed constitute *white space*. White space serves only to separate tokens that would otherwise be merged together.

Comments

The comment syntax is identical to the syntax of C and Golang. Both // to end-of-line and /* to */-style comments are supported. /* */-style comments are replaced with a single space during lexical analysis, so they separate tokens. //-style comments end with a newline, which naturally separates tokens.

Identifiers

```
identifier = letter { letter | acscii_digit } .
```

All programmer-defined variable and function names are called identifiers. Identifiers must start with a letter that may be followed by 0 to 14 letters or numbers. The underscore character “_” is defined as a lower-case letter .Implementation note: the lexical analyzer should enforce this 15 character limit on identifiers. Some identifiers are predeclared.

Keywords

The following keywords are reserved in YAPL. Some are not implemented in this first version of the language. Note that some keywords include the hash character (#) which is not a legal identifier character. The compiler handles this as a special case.

```
break        default      func         select
```

```
case          struct        #file         var
else          goto          #line         switch
const         if            #if           #else
#endif        continue      for           return
```

Operators and Punctuation

The following character sequences represent operators or punctuation in YAPL:

```
+     &      &&     ==     !=     (     )
-     |      ||     <      <=     [     ]
*     ^      ~      >      >=     =     ,
/     sizeof       <<     @      {     }
%     >>     !      .      ->     ~
```

Integer Literals

Numerical values may be given in hexadecimal (0xDDD..), octal (0oDDD…), decimal (DDD…), or binary (0b…). Hexadecimal values may contain the digits A through F, with uppercase encouraged but not enforced. Octal strings may not contain the digits 8 or 9. Binary strings may contain only 1 and 0. Any number may contain underscores for clarification. The underscores have no significance to the value.

Integral literals must be expressible in 16-bits without overflow..

Floating point and similar types like double and complex are not supported in YAPL. Library support for such types is enabled by the **`block`** types defined later.

```
int_lit         = decimal_lit | binary_lit | octal_lit | hex_lit .
decimal_lit     = "0" | ( "1" … "9" ) [ [ "_" ] decimal_digits ] .
binary_lit      = "0" ( "b" | "B" ) [ "_" ] binary_digits .
octal_lit       = "0" [ "o" | "O" ] [ "_" ] octal_digits .
hex_lit         = "0" ( "x" | "X" ) [ "_" ] hex_digits .

decimal_digits = decimal_digit { [ "_" ] decimal_digit } .
binary_digits  = binary_digit { [ "_" ] binary_digit } .
octal_digits   = octal_digit { [ "_" ] octal_digit } .
hex_digits     = hex_digit { [ "_" ] hex_digit } .
```

Char Literals

Single quotes ‘ introduce and terminate a char literal. Char literals may be defined by single ASCII characters, single character escapes, or hexadecimal escapes. Char literals define unsigned 8-bit values

Hexadecimal escapes consist of a backslash (\) followed by a lowercase x followed by two hexadecimal digits:

```
'\x00'     nul char
'\x04'     ASCII EOT (Ctrl-D)
'\xFF'     not an ASCII character (uint8 0xFF)
```

The single character escapes include:

```
\0    '\x00' nul char
\a    '\x07' alert or bell
\b    '\x08' backspace
\f    '\x0C' form feed
\n    '\x0A' line feed or newline
\r    '\x0D' carriage return
\t    '\x09' horizontal tab
\v    '\x0B' vertical tab
\\    '\x5C' backslash
\'    '\x27' single quote
\"    '\x22' double quote
```

Other escape-like sequences are illegal. Examples:

```
'\''          // byte literal containing single quote character
'aa'          // illegal: too many characters
'\k'          // illegal: k is not recognized after a backslash
'\xa'         // illegal: too few hexadecimal digit
```

String (byte array) literals

As a concession to modernity, string literals are encoded in UTF-8. String literals define byte arrays.; there is no string type in YAPL. String literals may contain literal UTF-8 characters (e.g. `"日本語"`) or hexadecimal escapes that encode them. Thus, the byte length of a string literal may be significantly greater than the number of characters. Since strings define byte arrays, the bytes of a string may be iterated using array indexing. Iterating the characters of a UTF-8 string is reserved to a library function not specified here. MOVE THIS: Unlike the C language, the address-of operator is mandatory when taking the address of an array. Attempting to assign an array to a variable results in a compile time error.

Source Program Structure

Source programs are constructed from one or more source files, called *compilation units*. This language system has no object file linker. The compiler generates Assembly Language source code. Complete source programs must be given to the compiler, which generates Assembly Language programs that are presented to the assembler as a unit. The assembler generates absolute binary executable files.

Despite the simple-minded nature of this process, the language defines the source file as a compilation unit. Compilation units affect the scope of variables as described in the next section.

Scope

The visibility of an identifier is called its scope. The scope of a declared identifier is the extent of source text in which the identifier is defined.

Identifiers defined at the compilation unit level, that is, not nested within any block, are visible throughout the compilation unit and are visible throughout the program if they begin with a capital letter. Such variables beginning with a capital letter are called *global* variables and such variables beginning with a lowercase letter are called *static* variables. The use of standard prefixes and suffixes is strongly recommended. Recommended prefixes and suffixes are **`g_`** for globals, **`s_`** for statics, and **`_t`** for types. Identifiers with the prefix **`N_`** are reserved to the compiler and must not be used in programs.

Namespaces

All identifiers not defined within a block, including structure tags, predefined type names, function names, and variables defined outside of any block, are taken from a single namespace called the global namespace and must not conflict. This difference from the C language allows structure tags to be used as type names without introducing syntactic ambiguity. In YAPL, one may write

```
struct atype_t {
    int n;
};
var atype_t s_foo;
```

That is, structure tags are types.

Within each structure type, the field names constitute a distinct namespace, That is, identifiers of field names may duplicate the names of identifiers from the global namespace without conflict.

Within the global namespace names must be unique.

Constants

Constant values result from the compile-time evaluation of constant expressions. Constant expressions have default types from the set of types described below. The default type of an expression results from the types of the literals and from the behavior of the operators used in the expression. Constant identifiers may be created from constant expressions using the **`const`** keyword, which specifies the type and name of the identifier. The compiler attempts to avoid allocating space in data memory for the values of constant identifiers by e.g. interpolating the values as the operands of immediate instructions. Char literals may be used to initialize uint8 or uint16 variables or fields

Variables

A variable names a storage location that holds a *value*. The set of permissible values is determined by the variable's *type*. Variables must be declared before they can be used**.** A declaration reserves a space in memory for a variable and specifies its type. Declarations begin with the keyword **`var`** followed by the type, name, and an optional initialization expression.

Types

Integral and reference types

YAPL's type system is simple. The language includes the machine types **`uint8`,** also called **`byte`**, **`int16`**, and **`uint16`** with the obvious meanings, plus the reference (pointer) types **`@byte`**, **`@int16`**, and **`@uint16`**. The types **`byte`**, **`int16`**, and **`uint16`** are known as *integral types* while **`@byte`**, **`@int`**, and **`@uint16`** are known as *reference types*.

The language also includes the types **`block32`**, **`block64`,** and **`block128`** which define 32-bit, 64-bit, and 128-bit otherwise-untyped storage areas respectively along with the corresponding reference types **`@block32`**, **`@block64`,** and **`@block128`.** The only operator defined on the block types is the address-of operator **`@`.** Aside from having their address taken, blocks may not participate in expressions. The block types are intended to support library implementations of other types, like long, float, double, UUID, and other user-defined types. Blocks are always aligned on 32-bit boundaries in memory.
These are all the basic built-in types. New types may be introduced by creating structures and arrays of these types.

The type names are predefined identifiers,, not keywords. The set of predefined types may be extended only by defining structures and arrays. There is nothing like the C language **`typedef`** or the Go language **`type`** keyword.

Booleans

Values are considered true if they are non-zero and false if they are zero. Thus all values can be considered booleans and can participate in boolean expressions. Evaluation of a comparison operator (<,>, <=, >=, ==, !=) produces the value 1 if the result is true or 0 otherwise.

Structures

A structure (“struct”) is a sequence of named elements, called fields, each of which has a name and a type.Fields must have predefined types or be arrays of predefined types having constant length. The compiler assigns successive storage locations for each field except when alignment restrictions require padding as described below.

Fields are extracted from a named structure variable using the dot operator (“.”) as in C. Fields are extracted from a reference to a structure using the arrow operator (“->”).

Structures are aligned on 16-bit boundaries unless they contain one or more block fields, in which case they are allocated on 32-bit boundaries. All reference types and all 16-bit integral types are allocated on 16-bit boundaries. Instances of all block types are aligned on 32-bit boundaries. Alignment rules may cause unallocated spaces in structured declarations. For example, the structure

```
struct example {
      byte b;
      block32 bl32;
}
```

Will be allocated on the next 32-bit boundary in memory, possibly leaving unused bytes, and will have three bytes of unused space after **`b`**. Arrange structure members from largest to smallest to avoid wasted space. Dynamic allocators must return storage aligned on 4 byte boundaries in memory.

The identifier after the keyword **`struct`** defines the type and may be used to define reference types or arrays:

```
struct struct_t {
      byte b;
      block32 bl32;
}
```

```
var struct_t exstruct, @eptr = @exstruct;
```

Arrays

A raise may be declared using the conventional syntax with square brackets. Examples: var b**yte[constant-expression]**, **var @byte[constant-expression]**, **struct_t[8]** etc. Only single-dimensional arrays are supported by the language. Multi-dimensional structures may be constructed by allocating single dimensional arrays of reference types and then dynamically allocating the space they point to.

Type Conversion

Types may be coerced using the type name as a conversion function name as shown in the following examples:

```
var int i = -1;
var uint16 u = uint16(i);
```

or

```
struct some_t  {
    byte b;
    int16 i;
}

var some_t @aPtr = @some_t(allocate(sizeof(@aPtr));
```

Reference types may not be assigned to integral types without generating a warning even with explicit type coercion.

```
// valid syntax, but illegal and generates a warning:
var uint16 anIntegralValue = uint16(aPtr);
```

There are no automatic or implicit type conversions nor promotions in YAPL, not even widening a **uint8 (byte)** to a **uint16**. Types must match strictly; all type conversions require explicit coercions.

Expressions

Operands

Operands denote the elementary values in an expression. An operand may be a literal, a (possibly qualified) non-blank identifier denoting a constant, variable, or function, or a parenthesized expression.

```
Operand     = Literal | OperandName | "(" Expression ")" .
Literal     = BasicLit | CompositeLit | FunctionLit .
BasicLit    = int_lit | float_lit | imaginary_lit | rune_lit |
string_lit .
OperandName = identifier | QualifiedIdent .
```

Constant Expressions

Constant expressions are evaluated within the compiler. They are composed of numbers, constant symbols previously defined in source code lexical order, and all the operators with the usual meanings.The precedence of operators is defined in the table below. Operations may be performed on integral values, array values, reference values and previously defined constant symbols. This evaluation must produce the same result as would be produced if all constants were sign extended to infinite bits and the expression was evaluated on a computer with infinite bit width.

Quoted strings are constant expressions.

Nonconstant expressions are evaluated at runtime and are called runtime expressions.

Runtime Expressions

All expressions not evaluated at compile time are called runtime expressions.They result in the generation of code to evaluate the expression. Runtime expressions allow the same set of operators having the same precedence as constant expressions. Previously-defined constant symbols may participate in runtime expressions..

Summary of operator precedence

Operators	**Associate**	**Meaning**	**Precedence**
parens () sizeof @ -> []	right to left	“@” like C ”*”. Others like “C”	highest (7)