Making a Programming Language From Scratch/Printable version

Making a Programming Language From Scratch

Contents

Decisions

Introduction

You are aiming to make a programming language. However, there are many existing languages in the market. What is the purpose of yours? Well, it is mostly about the minute differences in the syntax and the output code that creates the resultant diversity of languages. Mostly it is about memory format and speed of execution. However, generally these two cannot exist together. Thus there are to be certain tradeoffs that in the end will determine the importance of your language and where it stands in the global market. There are some minute factors such as general scope of variables, redundancy of instructions, style of compilation and execution speed differences, which may seem trivial at first but when they add up in a long source file, they can mean the difference between the program execution time in minutes and hours. These minute factors remain the driving force in the continuing research for new and better programming languages. The following sections will cover some major decisions you have to undertake before you even start on your language

Base Languages

It is very difficult to compile to machine language, and generally the next best step, assembly language is undertaken. However, even in assembly language, the actual file input and output operations necessary for compilation are very complex and difficult to write. However, fortunately there exist some high-level languages in which to write our compiler. Note that the selection of the Base language is not important for the end output quality, but for the compilation speed and ease of writing the compiler. This project is likely to be very long and thus it is imperative that you select a language in which you are comfortable.

Usually execution in compiled languages such as C++ and C is much faster than in Interpreted Languages. However, they have the disadvantage of having lower portability, and thus the choice of the Base language depends totally on your priorities.

Memory Format

Memory nowadays is very high, (usually around 4 GB RAM) , thus the earlier unthinkable format of default static memory layout can be considered. In this format, the variables declared within a function remain active, i.e. they retain their value even after the function returns control, up until the termination of the program. This means that functions can hold their values for longer, an important fact that can allow for much shorter and efficient programs. However, most languages have what is called 'Stack' variables, where variables are pushed onto a memory unit called stack , and they remain there until the function ends. However, after their life ends, the stack is not cleared but left as it is , just that now it is available to further variables who overwrite the previous variables. However, this leads to 'Garbage values' which can lead to program errors. Also, it demands a complicated system of data management.However, stack variables occupy much less space and are also accessed faster. Further, there is the option of register variables, which remain in registers which are accessed up-to 30 times faster than stack variables. However register variables require huge levels of complex data management which would make the compiler more complex.

After that, there is the option of .[MODE] directive in assembly. This depends on the type of processor you are writing the compiler for. For 808386 processor use .386 for 808486 use .486 and so on. Some assembly languages support multiple modes for one processor, however each mode is best suited for its own specific processor. Further more , there is the MODEL directive , namely FLAT model, BIG, TINY, LARGE and

MEDIUM. Each is best suited for its own type. In this book we use the FLAT model. A lot depends also on the MODE of the OS. Here we use the protected mode which provides 16 TB (can vary from machine to machine) of "virtual memory".

Moreover, in the case of actual memory allocation, there is NEAR and FAR pointers used for NEAR and FAR model respectively. In protected mode, the OS assumes NEAR mode and thus NEAR pointers.

Another thing to consider is the size of your variables. Commonly three variables int, char and float are used. In C compilers for 32-bit environment , int takes 4 bytes, char takes 1 byte and float requires 8 bytes (for double). There are further directives short, long, single and double which you may or may not include in your programming language.

Most assemblers handle memory allocation on their own, however with some you have to be more explicit and give precise instructions as to where exactly you want your variables to be.

Speed

Every language-maker knows that by far the most important factor distinguishing programming languages in these days is the speed of execution. However, speed does not come easy and if you want speed , you better be prepared for many days of hard labor in the process of what is known as optimization. Moreover, speed often clashes with memory layout, thus you need to decide which is more important to you. Redundancy is another factor, but to remove redundancy you need to write very long algorithms that check for every possibility of redundancy.

Line by Line Input System

Processing the File

To make a compiler obviously you have to take input from a file.Thus you need to be familiar with File I/O in at least one language you are comfortable in. Now how do you get the statements? Each line in a programming language is terminated by a very specific character which is NOT a newline. Most generally it is a semi-colon(;) or an opening brace({). So to get a complete line we only need to input up till the terminating character. Note that this approach means that you cannot have the terminating characters within even a string or a character, but that they have to be represented by their ASCII code.

Algorithm For Input

```
1.Input Character
2.Check if Character is terminating character
 2.1.If true , add character and terminate process
 2.2.Else add character and continue
 2.3.If line contains EOF(end of file) character, terminate program after processing line. [Usually done by the means of flags]
3.Repeat from step 1
```

This algorithm will take input from the file and then store it in the character array assigned to it.

Processing Keyword

Now, once you have the complete logical line, what are you going to do with it? Obviously you are going to process it. But how? There are two methods to it.

1. Lexing and Parsing the line
2.Getting keyword and taking appropriate action thereafter

In method 1 we take the line, run it through what is known as a 'Lexer' which breaks it down into tokens a 'Parser' which analyses it. Based on this it makes a wise decision and sends it to be processed accordingly.However this approach needs a very sophisticated system.

This book seeks to skip the processes of Parsing and Lexing and go straight into processing. Thus this book uses the Pseudo-Parser and Pseudo-Lexer approach (Method 2) which Lexes and Parses the statement differently for each type of statement, which allows for simpler compilers and lot of grammatical freedom.

The algorithm for this approach is rather simple:

```
1.Take character from character array storing logical line till we encounter a 'delimiter' like ',",[space] and {
2.Compare the keyword thus derived with the set of all defined keywords.
 2.1.If one matches, send the line to the matching function to process it.
 2.2.If none match, produce an error message
```

After this process is over, check for flags set by the line-input system, if the flag is set, terminate program.

Simple Data Types

Simple Data Types

What does 'Simple Data Types' mean? The term refers to data types such as float, int and char declared in single format. Thus this type of data declarations exclude arrays and structures which are covered later.

Format for Simple Data Declarations

```
[further definitions] type of variable variable name 1 [,(or);] [variable name 2] ...
```

```
[further definitions] can be:
short
long
single
double
static
local
extern
```

```
type of variable can be:
float
int
char
```

```
variable name can be any ASCII character(in some languages Extended-ASCII or UTF-8) beginning with an
alphabet(A-Z)(a-z)
```

Assembly equivalents for common data types

Assembly language , however does not recognize these data types. In fact, there is no distinction between an int and a char, or even between an int and a float (just that float variables are stored in a different format).

Assembly language does not recognize static, local or extern variables. All variables declared in the BSS or DATA section are static and all variables declared with Macro LOCAL are local. Local variables are declared in CODE sections

Also you cannot declare more than one variable in one logical line. Assembly language recognizes the following type of variables:

```
BYTE(equivalent to char)(takes 1 byte)
WORD(equivalent to short int)(takes 2 bytes)
DWORD(equivalent to int)(takes 4 bytes)
QWORD(equivalent to long int)(takes 8 bytes)
TBYTE(equivalent to long long int)(takes 10 bytes)
```

```
REAL4(IEEE format)(equivalent to single float)(takes 4 bytes)
REAL8(IEEE format)(equivalent to double float)(takes 8 bytes)
REAL10(IEEE format)(equivalent to long double float)(takes 10 bytes)
```

The first group is the integer group which store non-fractional data
The second group is the fractional group which stores fractional data

Algorithm

```
1.Scan for identifiers
 1.1 If "static" set static flag 1
 1.2 Else static flag 0
2.Scan for data type
 2.1 If "int" set int flag 1
 2.2 If "float" set float flag 1
 2.3 If "char" set char flag 1
3.Until ',' or '=' or ';' is encountered, get character and store in name array
 3.1
  3.1.1. If Float flag set write name and "real8"
  3.1.2. If Int flag set write name and "dword"
  3.1.3 If Char flag set write name and "byte"
 3.2
  3.2.1 If '='
   3.2.1.1 Until ',' or ';' is encountered get character and store in value array
   3.2.1.2 Write value after type
  3.2.2 If ';' terminate process
4 Repeat step 1
```

Sample Conversions

Original Code:

```
int a,b,c;
```

Resultant Code:

```
a dword ?
b dword ?
c dword ?
```

Original Code:

```
int a=1,b=2,c=3;
```

Resultant Code:

```
a dword 1
b dword 2
c dword 3
```

Note: All uninitialized variables are to be declared in the .BSS or .DATA? section while all initialized variables are to be stored in .DATA section. All local variables are to be declared in .CODE section in the following format

```
LOCAL [name of variable] [type of variable]
```

Arrays

Array Declarations

Array declarations are generally done along with all other simple variables. The format for array declarations is essentially an extension of the format for declarations of simple variables.

Format

```
[format as for simple declarations][,][name of variable]['[' or ',' or ';' or '='][if '[' size of array
(constant)][']'][value of array][next declaration]
```

Example:

```
int a,b,arr[5],arr2[5]=(1,2,3,4,5);
```

(Note that as we have designated '{' as terminating character we replace the common '{' with '('. If you have not done so you can also use the brace.)

Algorithm

This algorithm is the continuation of the previous algorithm, with the following steps to be added to check for array variables and to deal with them properly.

```
1. Get name.
2. If next character be [ then until the character be ] input character and store it in the index variable .
3. If next character be , or ; then write to data? section :
  [Name] [index] dup ?
4. Else add one character and then get character until character be ).
Then write to data section:
  [name] [value]
```

The dup keyword will replicate ? or 0 value for all the members without needing to individually initialize each.

Special for strings

Strings are initialized differently form other array variables in that they can be declared simultaneously in one chunk separated by "and" Also strings always end with char 0 (or '\0'). The following algorithm is to be appended with the previous one.

Algorithm for strings only:

```
1. if character after char array declaration not be'=' continue with rest of parent algorithm.
2. add one to index.(skip ")
3. while character not " get character and store in array.
```

```
4. write to .DATA section:
   [name] byte "[value]",0
```

Note that some assemblers may impose a limit on the actual size of initialized string (in MASM 6.1 it is 255 bytes or 255 individual characters). Note that the size of the initialized string is only that of the value provided, i.e. in the following example

```
char str[50]="hello";
```

the size of the array str is only 5+1 or 6 characters.

Array Referencing

An array has to be referred to in order for it to be useful. An array is referenced in the following format.

```
[...expression][array name][index of variable][...expression]
```

However assembly language does not accept this format. Moreover it takes the index as the number of bytes after the starting address, rather than the number of variable after the starting address.

Format in assembly

```
[...instruction][array name][number of bytes after starting][...instruction]
```

Further more the index cannot be a memory variable, but has to be a register or a constant.

The solution is the following set of instructions:

```
mov ebx,[index variable(can be a constant also)]
[assignment instruction to register of type of array][array name] /[ebx * type array]/
[assignment instruction to arr[ANUM] (increment ANUM) from register used above]
```

the '/' signify differential use of '['(here the brackets after / are to be copied into code)

Algorithm for referencing(upon detection of array variable or to be done as soon as instruction comes):

```
1. While character not '[' and not ';' and not '{'
   increment index.
2. If character be ';' or '{' end process.
3. While character not ']' get character and store in array.
4. Get name of array.
5. Use format as given above.
6. Replace reference by arr[ANUM-1]
7. Repeat step 1
```

Pointers

Introduction to pointers

Pointers are variables that store the addresses of other variables. This variable is generally a DWORD or a 32-bit integer, however the variable the address of which the pointer is storing can be of any type. Pointers are used extensively to pass static or local variables to be edited and are the major type of arguments provided to a void function.

Pointer Format

The following is the traditional C and C++ format to declare pointer variables.

[type of variable of which the pointer is storing value of][*(represents pointer)][variable name][initialization of pointer][,|;]

Note that pointer variables can be declared among the variable type of which it is storing the value. Example:

```
float f1,f2,*p1,*p2=&f2;
```

The & operator provides the address of the variable rather than the value stored in it.

Algorithm for declaration

The algorithm for this is as follows:

```
1. If character be * then go through following steps else skip it.
2. Get name of variable.
3. If next character not be = then write to .data section as [name of variable] ptr [type] ?
4. Else get value. Remove first character as  it will be &. Write to .data? section as [name of variable]
ptr [type] [value].
5. Repeat step 1.
```

Structures

Structural declarations

Structures are groups of different types of data in continuous locations. It is necessarily the grouping of different types of data together.

Format of structural declarations

```
Defstruct [Name of structure]{
.

.

.

}
```

Format of assembly:

```
[Name of structure] struct starts
.

.

.

[Name of structure] ends
```

Algorithm to process

```
1.If keyword defstruct then
2.While char not { get char and store in array name
3.Write to output file format as given above.
```

[note that choice of keyword is totally up to you.]

Structure variable declarations

All structures are useless unless they have some representatives in the physical memory. This is called a structure variable.
A structure variable is defined along with other variables with the *important difference that structure variables are not defined during initializing (at least in assembly)*. Thus we will deal with only uninitialized structure variables.

Format as of assembly:

```
[structure variable name] [parent structure name] <?>
```

The <?> is the structure dup operator which sets value 0 or 0.0 to all sub-components

Algorithm

```
1.If keyword struct then
2.Scan parent structure name
[Duplicate algorithm for getting simple variables]
3.Write to file in format as given above.
```

Sample Conversion

In HLL

```
defstruct boy{
 int age,grade;
 float marks;
 char name[20];
}
```

In Assembly

```
boy struct starts
age dword ?
grade dword ?
marks real8 ?
name byte 19 dup (?),0
boy ends
```

Simple Expressions

Expressions

An expression includes all forms of commands that require usage of the ALU excepting logic functions . They also include function calls but these will be covered later. In this chapter we deal with the conversion of simple expressions or expressions excluding parenthesis and function calls or any sort of referencing.

Basic Operations

We all know the four basic arithmetic operations that is +,-,*,/. Together they combine to form expressions such as : a*b+c

However assembly language does not recognize such expressions, but it only recognizes binary commands such as: add a,b
mul b,c

Also it does not recognize precedence. It just executes commands as they are presented in a sequential order. Thus we have to order theses commands by ourselves.

Also the results are always stored in a certain location known as a register. More specifically it is stored in the Eax register.

Instructions

There are the following instructions for each operation on integers. For operations on char replace eax by al:

Assignment

```
Mov [operand1],[operand2]
```

Add

```
add [operand1],[operand2]
```

Subtract

```
sub [operand1],[operand2]
```

Multiplication

```
Mov Eax,[operand1]
Imul [operand2]
```

Division

```
Cdq (sign extension, mandatory but only once in the program)
Idiv [operand1],[operand2]
```

Modulus

```
Cdq
Idiv [operand1],[operand2]
Mov Eax,edx
```

Increment

```
Inc [operand]
```

Following are the corresponding instructions for floats.

Assignment

```
Fld [operand2]
Fstp [operand1]
```

Addition

```
Fld [operand1]
Fadd [operand2]
```

Subtraction

```
Fld [operand1]
Fsub [operand2]
```

Multiplication

```
Fld [operand1]
Fmul [operand2]
```

Division

```
Fld [operand1]
Fdiv [operand2]
```

There is no modulus for float variables.

Algorithm For Conversion

Part1 Multiplication,division,modulus

```
1. While character not ; or *,/,%.
   increment index
```

```
 1.1 If character be ; goto part2
 1.2 get operand before operator and after
 1.3 depending on operator use the appropriate instructions as given above.
 1.4 remove the operands and operator from the line
 1.5 if next operator be add or sub replace by var[NUM] where NUM is an int and incremented. Write var[NUM]
as a variable of type of expression. Assign eax to var[NUM].
 1.6 else replace by eax.
2. Repeat step 1
```

For char replace eax by al and for float assign st(1) to var[NUM].

Part2 Addition,subtraction

```
1. While character not ; or + or -
   increment index
 1.1 If character be ; end process
 1.2 get operand before and after the operator
 1.3 depending on operator use the appropriate instruction as given above
 (Follow steps 1.4 to 1.6)
2. Repeat step 1
```

Complex Expressions

Parenthesizes

The previous chapter deals with the processing of expressions devoid of parenthesizes. However in real-world model, usually more than one level of expressions are used, and that obviously means the usage of parenthesizes.

Syntax

```
...Expression...(...Expression...More parenthesizes...)...
```

Algorithm for dealing with Parenthesis.

```
1.Go through the entire line. Increment index of array until element is ';' or ')'.
(This assumes that the expression has gone through function call processing, which is dealt with later.)
 1.1 If ';' terminate process
 1.2 If ')' continue.
2. While element not '(' decrement index.
3. Remove segment thus isolated by '(' and ')'. Store in separate array. Replace by 'pa(pnum)'.
4. Replace ')' by ';' and append to start 'pa(pnum)=' (replace pnum by actual value of variable)
5.Send to previous algorithm.
Repeat step 1.
```

Sample Expression Conversion

```
i p=(p*r*t)/100;
```

the i signifies that it is going to be an integer expression.

Result:

```
Mov eax,p
IMUL r,eax
MOV eax,eax
MOV eax,eax
IMUL t,eax
MOV eax,eax
MOV va1,eax
MOV eax,va1
MOV pa1,eax
MOV eax,pa1
CDQ
IDIV pa1/100
MOV p,eax
```

Now, this isnt the best of results, but this result works. However, there are huge opportunities for optimization. As you can see, nearly a third of this code is garbage with meaningless instructions like:

```
MOV eax,eax
```

Other times there are cases that the compiler loads the value onto eax and then dumps it onto another variable. (On my system it was intentional as the instructions did not work with two memory operands.

Try to optimize this algorithm by yourself before moving forward.

Finally Some Advantages!!

Here we have an advantage over general market languages. Because they are made for portability , they usually use what is known as a 'Emulator' which does NOT use the math co-processor but uses the CPU to *integer emulate float calculations. In our case we use direct float commands, which reduces portability but increases execution speed by almost 100 times!!*

Comparing Two Values

Comparisons

Comparisons in programming are the logical comparison between two data types of the same type to determine which is larger, or which is smaller or if both are equal. Based on the result thus derived which is a Boolean value (1 or 0) certain statements are executed or not.

Whole number comparisons

Whole numbers include chars and ints of all types. The basic format for all comparisons is the same.

```
CMP [ operand1 ] [ operand2 ]
```

The conditional jumps based on this result are :

```
JL or jnge for lesser than
Jle or jng for lesser than equal to
JG or Jnle for greater than
Jge or Jnl for greater than equal to
Je for equal to
Jne for not equal to
```

Floatational comparisons

The syntax for floatational comparisons are more complex.

```
Fld [ operand1 ]
Fld [ operand2 ]
Fcompp
Fstsw ax
Sahf
```

Comparisons based on this are:

```
Jc for lesser than
Jce for lesser than equal to
Je for equal to
Jne for not equal to
Ja for greater than
Jae for greater than equal to
```

Logical appendations

Logical appendations are when two or more conditions are logically connected such that the result of one influences the result of the other.

For example: If(a>b&&b>c){ This condition depends on both the logical relations of a,b and c. If any one of the double conditions are false the resultant output is false.

The counterpart the || operator is used for or relations, i.e if any one of the double conditions are true then the resultant is also true.

Note that these logical operators can be used for more than two conditions.

The Braces Problem

The big bad braces problem

One of the most fundamental problems that I faced during the creation of my language was the braces problem. Thus I devoted this chapter to this problem and it's solution.

In most modern languages block statements are terminated by the braces{}. However, multiple statements share the braces, such as if,while,else,elseif,functions,e.t.c.

Thus the problem I faced was this: How do you recognize the end brace } as belonging to which statement. Remember that in the line by line input method we will only get "}" as our line. After thinking for many days I finally came upon a solution.

Solution

We need to open the generated output file in read more. We then search for a comment of format !!? [Block statement name]

And if that statement is not terminated by ??! [Block statement name]

Then it is our required statement. Note that the required statement is the **last** open statement.

We close this statement by the above mentioned format and also include in the code the necessary instructions.

Localizing

Functions and scope

Before we do anything with functions we must localize the variables. Note that in case of procedures variables are not local but are global.

Scope

Anything that is declared within a function remains within the function. This essential rule called the Scope rule has governed the creation of programming languages since the time of C.

In the case of automatic or local variables it is easy . All variables that are declared with the LOCAL keyword are by default limited to the procedure only. Thus parameters are also to be declared as LOCAL however this is already done by the PROC keyword

For static variables there is no such facility and thus the localizing must be done manually. One cheap and easy method is to simply append the name of the function to the variable. But this method requires that you must convert the name of each and every static variable every where it is used. The final algorithm totally depends on you.

Function Definitions

Function definitions

There is nothing called functions in assembly. However, there is a mildly resembling counterpart called procedures. However procedures do not return any values. This critical problem will be discussed later on.

Syntax of function definition using PROC

It is not a necessity to use the PROC keyword for all your function declarations. However it is much easier to do it this way instead of its counterpart method involving code segment pointers. The syntax for function definitions using PROC is as follows:

```
[ Func name ] PROC [param1] [param2]...
.
.
.
[ Func name ] ENDP
```

All functions have to be declared this way.

Function Call

Function calls

Functions need to be called if they are to be of any use. There are two types of calling a function.

Non void call: A non void call is used for all functions that return a value. These functions are called and the returning value used for expressions or for logical processing.