ترجمه از زبان سطح بالا به زبان ماشین و برعکس

TRANSLATORS

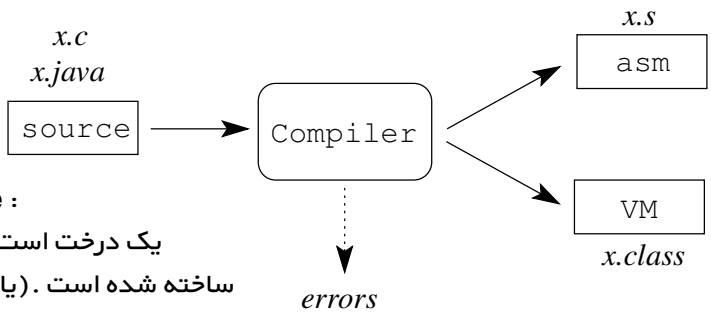
الزاما زبان مبدا سطحش بالاتر از زبان مقصد نیست
می تواند هم سطح باشد یا مبدا سطح پایین تری از مقصد داشته باشد

عملکرد کد نباید عوض شود

ورودی دیگر : باید گرامر را خبر داشته باشد

What's a Compiler???

- At the very basic level a compiler translates a computer program from source code to some kind of executable code:



abstract syntax tree :
یک درخت است که هر تکه کد طبق کدام قانون گرامر
ساخته شده است . (یا به آن داده میشود یا کامپایلر آن را میسازد)
برای بررسی این که جمله از لحاظ syntax درست هست یا نه ؟

- Often the source code is simply a text file and the executable code is a resulting assembly language program: `gcc -S x.c` reads the C source file `x.c` and generates an assembly code file `x.s`. Or the output can be a virtual machine code: `javac x.java` produces `x.class`.

What's a Language Translator???

- A compiler is really a special case of a language translator.
- A translator is a program that transforms a "program" P_1 written in a language L_1 into a program P_2 written in another language L_2.
- Typically, we desire P_1 and P_2 to be semantically equivalent, i.e. they should behave identically.

Example Language Translators

source language	translator	target language
LaTeX	latex2html ⟶	html
Postscript	ps2ascii ⟶	text
FORTRAN	f2c ⟶	C
C++	cfront ⟶	C
C	gcc ⟶	assembly
.class	SourceAgain ⟶	Java
x86 binary	fx32 ⟶	Alpha binary

Compiler Input

Text File Common on Unix.

Syntax Tree A structure editor uses its knowledge of the source language syntax to help the user edit & run the program. It can send a syntax tree to the compiler, relieving it of lexing & parsing.

Compiler Output

Assembly Code Unix compilers do this. Slow, but easy for the compiler. ورودی assembler

Object Code .o-files on Unix. Faster, since we don't have to call the assembler.

Executable Code Called a load-and-go-compiler.

Abstract Machine Code Serves as input to an interpreter. Fast turnaround time.

C-code Good for portability. Cfront : c++ -> c

obj_flle

یک مرحله فراتر از کد اسمبلی است . دیگر نیازی به
صدا زدن assembler نداریم زیرا کد به زبان ماشین را داریم
که سریع تر است .

Executable code :

در obj code فقط تکه کدی که به کامپایلر دادیم را تبدیل به صفر و یک کرده
و قابلیت اجرا شدن ندارد (زیرا ممکن است نیاز به کتابخانه خاصی داشته باشد که
آن هم باید تبدیل به صفر و یک شود یا بخشی از یک پروژه بزرگتر باشد)

Abstract machine code :

یک کدی است پیچیدگی های کد به زبان سطح بالا را ندارد
به سخت افزار هم بستگی ندارد . (java bytecode)

Profiler Information :

بهینه سازی انجام میدهد اما تفاوت دارد با بهینه سازی انجام شده در فازهای کامپایلر : در فاز بهینه سازی ، کامپایلر روی کد اجرا نشده بهینه سازی را انجام می دهد ولی در وظایف بهینه سازی در زمان اجرا انجام می شود . استفاده از اطلاعات آماری که بعد از اجرا به دست می آید و یکبار دیگر کامپایل می شود این بار از آن اطلاعات آماری استفاده میشود . شناسایی مسیر پرکاربرد و در نظر گرفتن متغیرهای آن مسیر در ثباتها که موجب بالا رفتن سرعت برنامه می شود.

Compiler Tasks

قوانین را بررسی کند که قابل نمایش با bnfنیستند

Static Semantic Analysis Is the program (statically) correct? If not, produce error messages to the user.

Code Generation The compiler must produce code that can be executed. کدی که پیچیدگی زیادندارد و وابستگی بیشتری به سخت افزار دارد

Symbolic Debug Information The compiler should produce a description of the source program needed by symbolic debuggers. Try `man gdb`.

Cross References The compiler may produce **cross-referencing** information. Where are identifiers declared & referenced?

Profiler Information Where does my program spend most of its execution time? Try `man gprof`.

Symbolic Debug Information :

کامپایلر هنگام کامپایل با یک سری متغیر روبه رو میشود و یک سری function را هم داریم

کامپایلر لزوما همان اسم های funcها را حفظ نمی کند و اسم ها را تغییر می دهدو اطلاعاتی به آن اضافه می کند و بعد آن را جایگزین می کند و آدرس نسبی داریم

کامپایلر این نگاشت ها را حفظ می کند . نگاشت این که یک متغیر به چه اسمی تغییر پیدا کرده است . برای اعلام یک مشکل از سمت debugger استفاده میشود

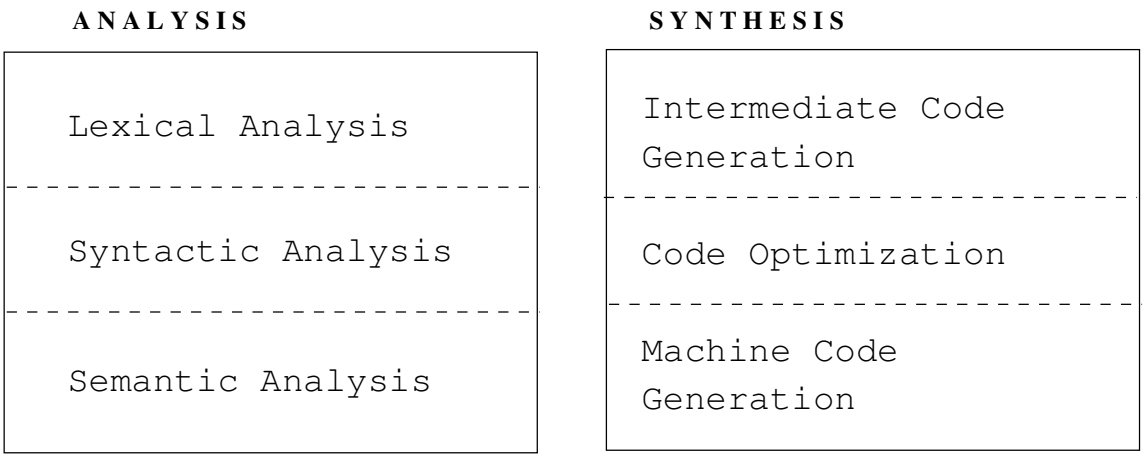
Cross Reference :

یک سری اطلاعات است که می تواند در قالب یک جدول که کامپایلر می سازد ، وجود داشته باشد . و بیان می کند که توابع توسط چه توابع دیگری صدا زده شده اند . این سلسله مراتب را حفظ می کند

Lexical. + Syntax + Semantic + Intermediate Code = Front-END

OPtimization + Machine code = backend

Compiler Phases



Lexical Analysis :

کلمه ها را کاراکتر به کاراکتر می خواند وقتی که تشخیص داد که کلمه تمام شده آن واحد را به عنوان token در نظر میگیرد و به سراغ کلمه بعدی میرود

Synatax Analysis :

بررسی این که token ها طبق syntax و گرامر درست در کنار یکدیگر قرار گرفته‌اند یا نه ؟ . ساختن AST و تکمیل در فاز بعدی

Syntax Analysis:

زمانی که یک گرامر را با bnf بیان می کنیم . یک سری قوانین امکان ندارد توسط این گرامر خودشان را نشان دهند مثل : یک متغیر قبل از اولین استفاده باید تعریف شده باشد یا یک funcقبل از صدا زدن باید تعریف شده باشد یا زمان انتصاب typeها باید همخوانی داشته باشد که این مسائل را باید کامپایلر جداگانه بررسی کند

Intermediate Code Genreation :

مثل abstract machine code می تواند به دست مفسر دهد یا خودش باز روی کد میانی کار کند و کار را جلو ببرد

تولید کد میانی به منظور بهینه سازی مورد استفاده قرار میگیرد (کدی زده شده که به سخت افزار بستگی ندارد).این کد می تواند در اختیار مفسر قرار گیرد

Code Optimization :

حذف متغیرهای بی استفاده یا کوتاه سازی محاسبات و ....

Multi-pass Compilation

- The next slide shows the outline of a typical compiler. In a unix environment each pass could be a stand-alone program, and the passes could be connected by pipes:

```
lex x.c | parse | sem | ir | opt | codegen > x.s
```

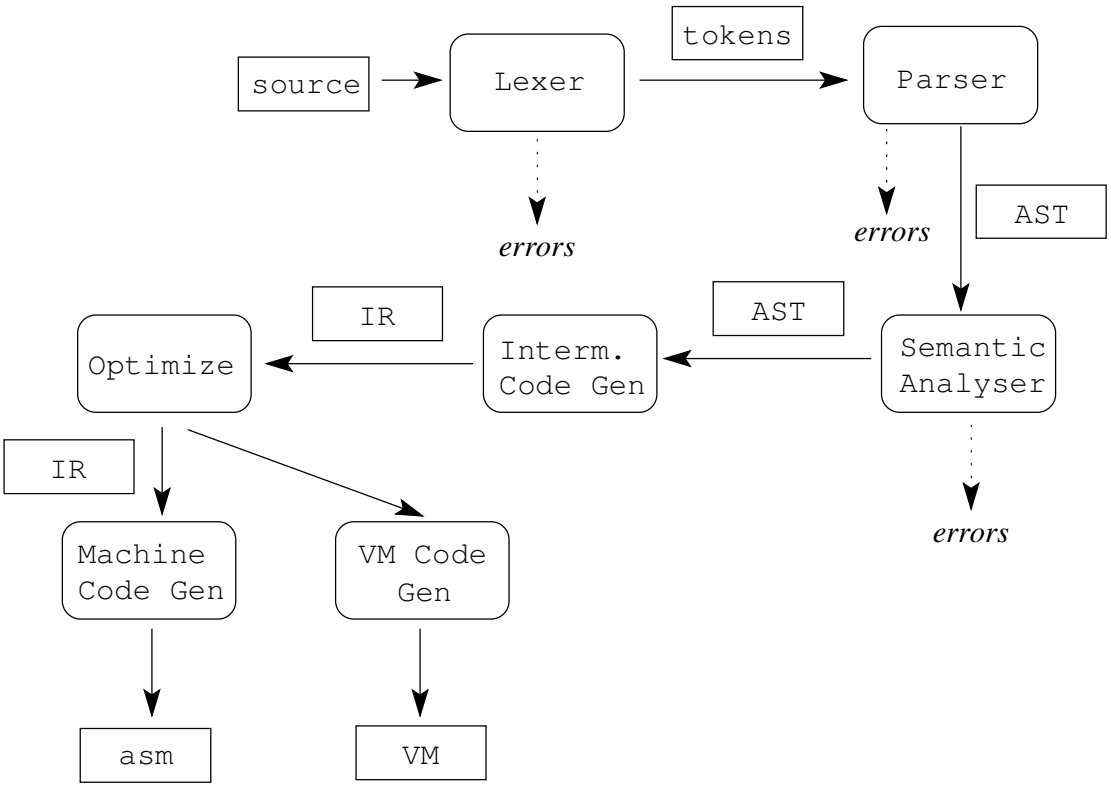
- For performance reasons the passes are usually integrated:

```
front x.c > x.ir
back x.ir > x.s
```

The front-end does all analysis and IR generation. The back-end optimizes and generates code.

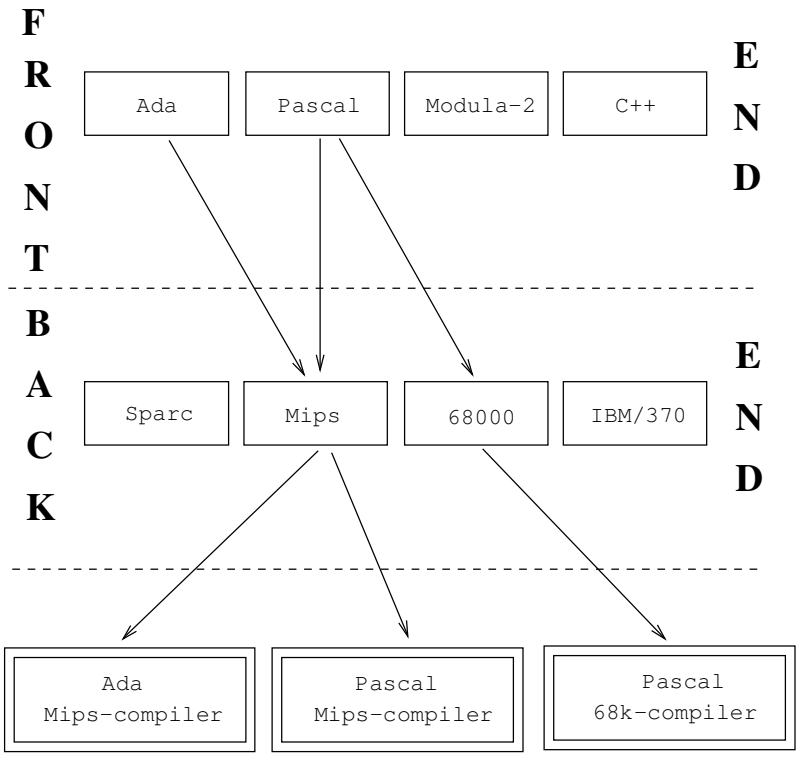
برای تولید فایل نهایی چند بار داده ورودی را می خواند و هر چه قدر زبان پیچیده تر باشد احتمال اینکهmultipass compiler داشته باشیم بیشتر است

Multi-pass Compilation...



یک تکینیک برای ساختن کامپایلر مورد استفاده قرار میگیرد :
کامپایلر های امروزی دیگر از صفرد ساخته نمیشوند!!
کامپایلرهای امروزی از کنارهم قرار دادن کامپایلرهای دیگر ساخته شوند .

Mix-and-Match Compilers



INTERPRETERS

مفسر خود یک برنامه است و درجا یک سری محاسبات را انجام می دهد. خط به خط برنامه را می خواندو آن را مانند یک cpu
اجرا می کند و درجا آن قدر متغیرها را دستکاری می کند تا به خروجی برنامه ما برسد
مفسر به هر زبانی که قابل اجرا باشد نوشته می شود معمولا از زبان سطح پایین تر از ورودی استفاده می شود.

خروجی آن : خروجی برنامه است و فایل به زبان سطح ماشین تولید نمی شود !!

Interpretation

- An interpreter is like a CPU, only in software.
- The compiler generates *virtual machine* (VM) code rather than native machine code.
- The interpreter executes VM instructions rather than native machine code.

چه حسنی دارند ؟

حجم فایل کمتری دارد .

Interpreters are در کار با مفسر کدی داریم که مستقل از سخت افزار است .ورودی یک مفسر مستقل از سخت افزار باشد

slow Often 10–100 times slower than executing machine code directly.

portable The virtual machine code is not tied to any particular architecture.

کامپایلر سریع تر می تواند عمل کند چرا ؟

مفسر هر خط را به دستورات ساده تر می شکاند و خودش آن را اجرا می کند و به سراغ خط بعد می رود. هیچ چیزی را نگه نمیدارد بنابراین وقتی به یک دستور تکراری برسد . دوباره آن را به دستورات ریز تر می شکاند و اجرا می کند . (هیچ چیز به ازای یک خطی که خوانده در حافظه خود نگه نمیدارد)
ولی کامپایلر هر خط را به صفر و یک تبدیل میکند و از آن به بعد هر چند بار که لازم باشد آن صفر و یک اجرا میشود.

کامپایلری که ورودی مفسر را فراهم می کند می تواند مستقل از سخت افزار کار خود را جلو ببرد و از آن جایی که وابسته می شود
همه چیز را به مفسر پاس دهد . پس می توانیم کامپایلر داشته باشیم روی سیستم های مختلف و کد virtual را بین سیستم ها منتقل کنیم
و هر سیستم با مفسر خودش آن را اجرا کند
هر چه قدر زبان ویژگی هایی داشته باشد که در زمان اجرا معلوم می شود ،بهتر است از مفسر استفاده شود .
هر چه قدر که ویژگی های static داشته باشیم می توانیم از کامپایلر استفاده کنیم .

Interpretation...

Interpreters are

slow Often 10–100 times slower than executing machine code directly.

portable The virtual machine code is not tied to any particular architecture.

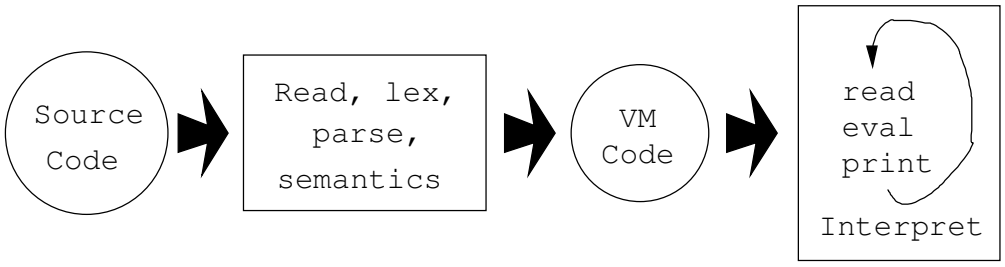
Interpreters work well with

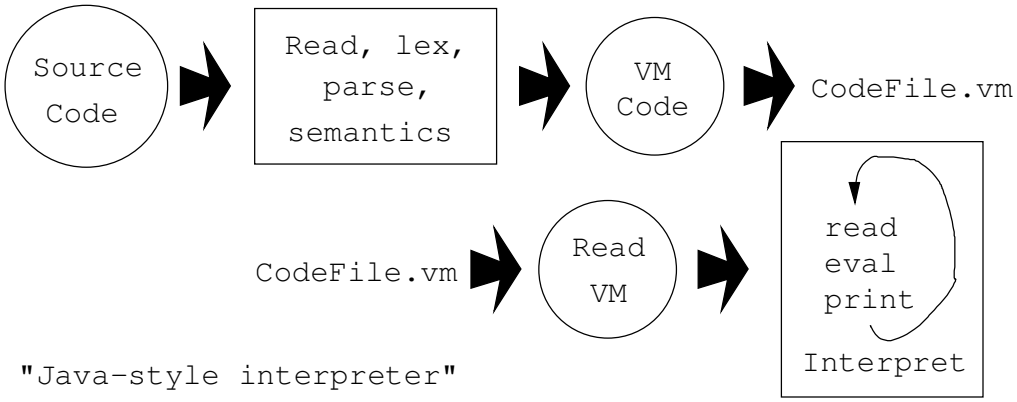
very high-level, dynamic languages (APL,Prolog,ICON) where a lot is unknown at compile-time (array bounds, etc).

Kinds of Interpreters

فایل به صورت دائمی در جایی ثبت نمی شود و این کد در حافظه موقت نگه داری می شوند

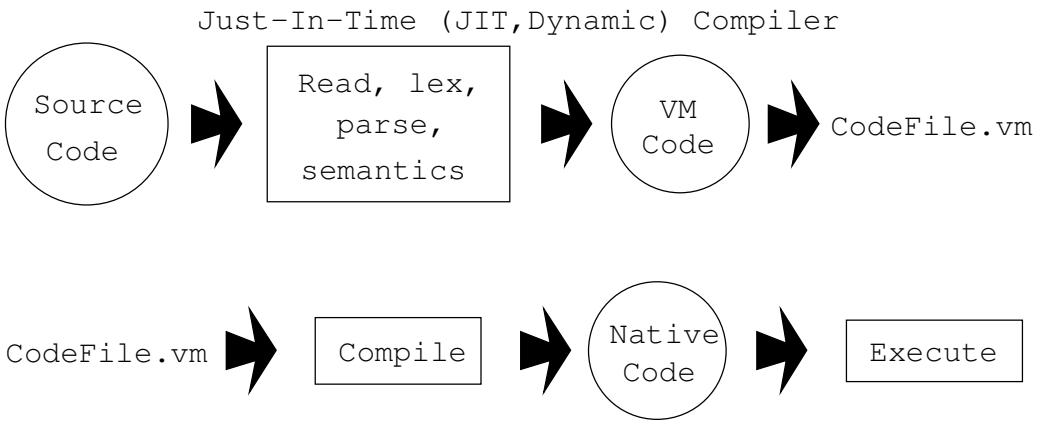
"APL/Prolog-style (load-and-go/interactive) interpreter"





"Java-style interpreter"

Kinds of Interpreters. . .



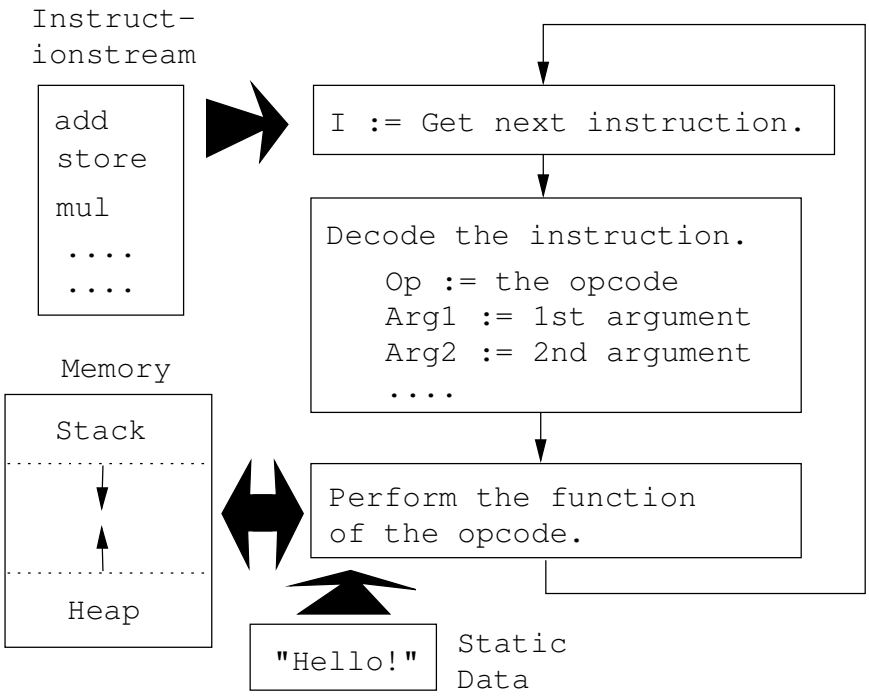
Actions in an Interpreter

- Internally, an interpreter consists of
  1. The interpreter *engine*, which executes the VM instructions.
  2. *Memory* for storing user data. Often separated as a heap and a stack.
  3. A stream of VM instructions.

stack vs heap :

حافظه ای که به صورت dynamic با آن کار کرد (heap)

Actions in an Interpreter…



Stack-Based Instruction Sets

- Many virtual machine instruction sets (e.g. Java bytecode, Forth) are *stack based*.

  add pop the two top elements off the stack, add them together, and push the result on the stack.

  push X push the value of variable X.

  pusha X push the address of variable X.

  store pop a value V, and an address A off the stack. Store V at memory address A.

Stack-Based Instruction Sets...

- Here's an example of a small program and the corresponding stack code:

Source Code	VM Code

`VAR X,Y,Z : INTEGER;`
`BEGIN`
`  X := Y + Z;`
`END;`
	`pusha X`
	`push Y`
	`push Z`
	`add`
	`store`

Register-Based Instruction Sets

- Stack codes are *compact*. If we don't worry about code size, we can use any intermediate code (tuples, trees). Example: RISC-like VM code with ∞ number of virtual registers $R_1, \cdots$:

add R_1, R_2, R_3 Add VM registers R_2 and R_3 and store in VM register R_1.

load R_1, X R_1 := value of variable X.

loada R_1, X R_1 := address of variable X.

store R_1, R_2 Store value R_2 at address R_1.

Register-Based Instruction Sets...

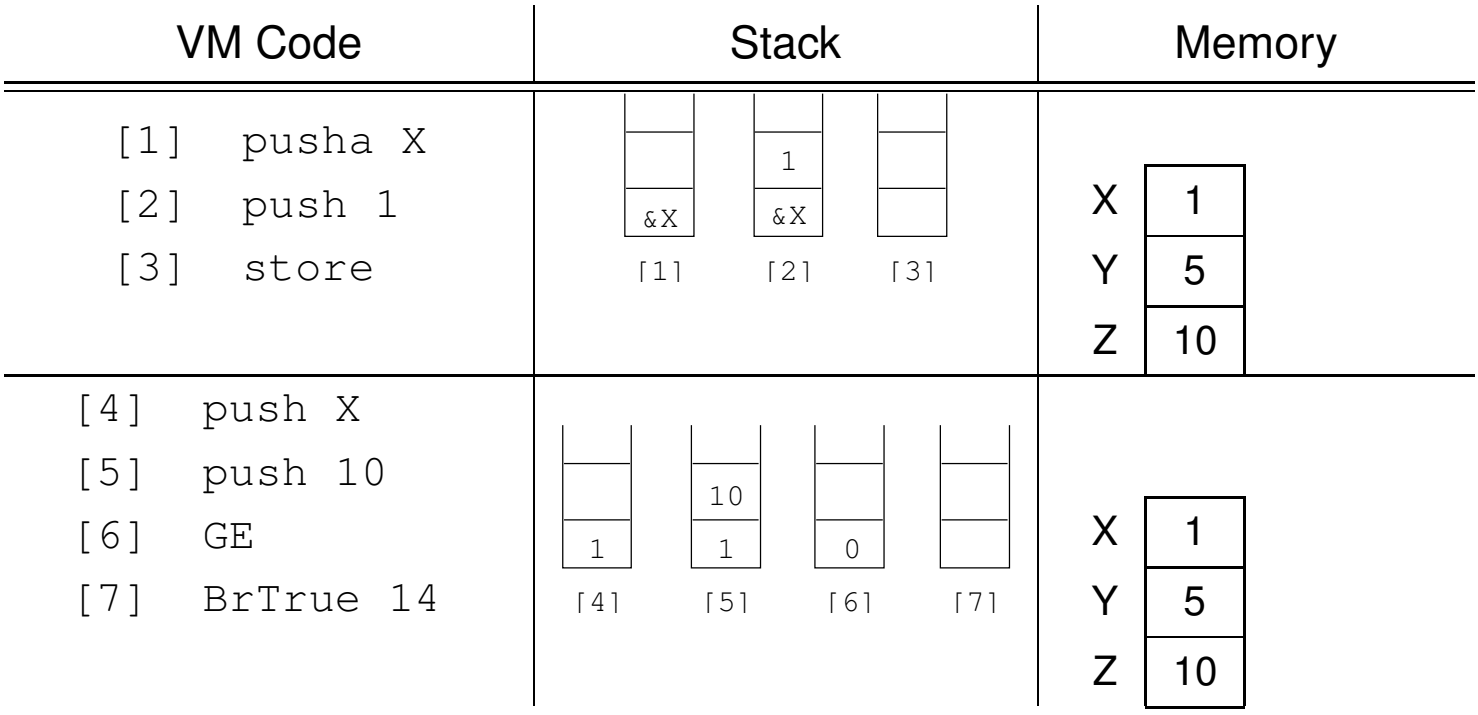
- Here's an example of a small program and the corresponding register code:

Source Code	VM Code
`VAR X,Y,Z : INTEGER;`	`load` R_1, Y
`BEGIN`	`load` R_2, Z
`  X := Y + Z;`	`add` R_3, R_1, R_2
`END;`	`loada` R_4, X
	`store` R_4, R_3

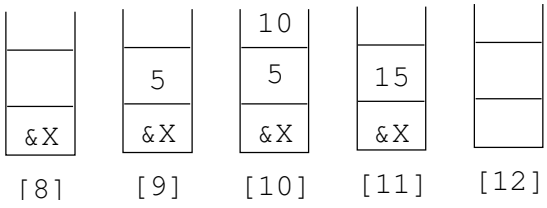
Stack Machine Example I

Source Code	VM Code
`VAR X,Y,Z : INTEGER;`	`[1] pusha X`
`BEGIN`	`[2] push 1`
`  X := 1;`	`[3] store`
`  WHILE X < 10 DO`	`[4] push X`
	`[5] push 10`
	`[6] GE`
	`[7] BrTrue 14`
`    X := Y + Z;`	`[8] pusha X`
	`[9] push Y`
	`[10] push Z`
	`[11] add`
`  ENDDO`	`[12] store`
`END.`	`[13] dump 4`

Stack Machine Example (a)



Stack Machine Example (b)

VM Code	Stack	Memory
[8] pusha X [9] push Y [10] push Z [11] add [12] store		X 15 Y 5 Z 10
[13] jump 4		