ICL Project
Mini Python

Interpretation and Compilation of Languages

Nova School of Science and Technology
Mário Pereira   mjp.pereira@fct.unl.pt

Version of April 21, 2025

The goal is to build a compiler for a tiny fragment of the Python language, called Mini Python in the following, to x86-64 assembly. This fragment contains Booleans, integers, strings, and lists. It is 99% compatible with Python 3. This means that Python documentation and a Python interpreter can be used as a reference when needed.

The syntax of Mini Python is described in Sec. 1. A parser is provided (for both OCaml and Java). You have to implement static type checking (Sec. 2) and code generation (Sec. 4).

1 Syntax

We use the following notations in grammars:

$\langle rule\rangle^\star$	repeats $\langle rule\rangle$ an arbitrary number of times (including zero)
$\langle rule\rangle^\star_t$	repeats $\langle rule\rangle$ an arbitrary number of times (including zero), with separator t
$\langle rule\rangle^+$	repeats $\langle rule\rangle$ at least once
$\langle rule\rangle^+_t$	repeats $\langle rule\rangle$ at least once, with separator t
$\langle rule\rangle?$	use $\langle rule\rangle$ optionally
($\langle rule\rangle$)	grouping

Be careful not to confuse "*" and "$^+$" with "`*`" and "`+`" that are Python symbols. Similarly, do not confuse grammar parentheses with terminal symbols `(` and `)`.

1.1 Lexical Conventions

Comments start with `#` and extend to the end of line. Identifiers follow the regular expression $\langle ident\rangle$:

$$
\begin{array}{lcl}
\langle \textit{digit} \rangle & ::= & \texttt{0–9} \\
\langle \textit{alpha} \rangle & ::= & \texttt{a–z} \mid \texttt{A–Z} \\
\langle \textit{ident} \rangle & ::= & (\langle \textit{alpha} \rangle \mid \texttt{\_})\ (\langle \textit{alpha} \rangle \mid \texttt{\_} \mid \langle \textit{digit} \rangle)^\star
\end{array}
$$

The following identifiers are keywords:

`and`	`def`	`else`	`for`	`if`	`True`	`False`
`in`	`not`	`or`	`print`	`return`	`None`	

Integer literals follow the regular expression $\langle \textit{integer} \rangle$:

$$\langle \textit{integer} \rangle \ ::= \ \texttt{0} \mid \texttt{1–9}\ \langle \textit{digit} \rangle^\star$$

String literals are written between quotes (`"`). There are two escape sequences: `\"` for the character `"` and `\n` for a new line character. We assume that string literals do not contain any character `\` beyond these two escape sequences.

In the Python language, block structure is defined by line indentation, *i.e.*, the number of spaces at the beginning of a line (it is assumed that files do not contain tabulation characters). Most of the work is done by the lexical analyzer, which produces `NEWLINE`, `BEGIN`, and `END` tokens corresponding to the end of lines and the increase or decrease in indentation respectively. The algorithm is as follows. The lexical analyzer maintains a stack of integers, representing successive current indentations. This stack is sorted, with the largest value at the top. Initially, the stack contains a single value, namely 0. When the lexical analyzer encounters a carriage return, it produces a `NEWLINE` token, then measures the indentation at the beginning of the next line, n, and compares it with the indentation at the top of the stack, m. There are three cases:

- if $n = m$, we do nothing;
- if $n > m$, we push n on the stack and we emit a second token `BEGIN` ;
- if $n < m$, then we pop until we find the value n, emitting a token `END` for each value strictly greater than n popped from the stack (the value n, if any, stays on top of the stack); if n does not appear on the stack, we fail with an indentation error.

In this process, empty lines, that is lines only containing spaces or comments, are ignored.

1.2 Syntax

The grammar of source files is given in Fig. 1. The entry point is $\langle \textit{file} \rangle$. Associativity and priorities are given below, from lowest to strongest priority.

operation	**associativity**	**priority**
`or`	left	lowest
`and`	left	
`not`	—	
`<` `<=` `>` `>=` `==` `!=`	—	↓
`+` `-`	left	
`*` `//` `%`	left	
`-` (unary)	—	
`[`	—	strongest

Note that, contrary to Python, an expression such as `x < y < z` is not part of our syntax.

$$
\begin{array}{lcl}
\langle \textit{file} \rangle & ::= & \texttt{NEWLINE?}\ \langle \textit{def} \rangle^{\star}\ \langle \textit{stmt} \rangle^{+}\ \texttt{EOF} \\
\langle \textit{def} \rangle & ::= & \texttt{def}\ \langle \textit{ident} \rangle\ \texttt{(}\ \langle \textit{ident} \rangle^{\star}_{,}\ \texttt{)}\ \texttt{:}\ \langle \textit{suite} \rangle \\
\langle \textit{suite} \rangle & ::= & \langle \textit{simple\_stmt} \rangle\ \texttt{NEWLINE} \\
 & | & \texttt{NEWLINE}\ \texttt{BEGIN}\ \langle \textit{stmt} \rangle^{+}\ \texttt{END} \\
\langle \textit{simple\_stmt} \rangle & ::= & \texttt{return}\ \langle \textit{expr} \rangle \\
 & | & \langle \textit{ident} \rangle\ \texttt{=}\ \langle \textit{expr} \rangle \\
 & | & \langle \textit{expr} \rangle\ \texttt{[}\ \langle \textit{expr} \rangle\ \texttt{]}\ \texttt{=}\ \langle \textit{expr} \rangle \\
 & | & \texttt{print}\ \texttt{(}\ \langle \textit{expr} \rangle\ \texttt{)} \\
 & | & \langle \textit{expr} \rangle \\
\langle \textit{stmt} \rangle & ::= & \langle \textit{simple\_stmt} \rangle\ \texttt{NEWLINE} \\
 & | & \texttt{if}\ \langle \textit{expr} \rangle\ \texttt{:}\ \langle \textit{suite} \rangle \\
 & | & \texttt{if}\ \langle \textit{expr} \rangle\ \texttt{:}\ \langle \textit{suite} \rangle\ \texttt{else}\ \texttt{:}\ \langle \textit{suite} \rangle \\
 & | & \texttt{for}\ \langle \textit{ident} \rangle\ \texttt{in}\ \langle \textit{expr} \rangle\ \texttt{:}\ \langle \textit{suite} \rangle \\
\langle \textit{expr} \rangle & ::= & \langle \textit{const} \rangle \\
 & | & \langle \textit{ident} \rangle \\
 & | & \langle \textit{expr} \rangle\ \texttt{[}\ \langle \textit{expr} \rangle\ \texttt{]} \\
 & | & \texttt{-}\ \langle \textit{expr} \rangle \\
 & | & \texttt{not}\ \langle \textit{expr} \rangle \\
 & | & \langle \textit{expr} \rangle\ \langle \textit{binop} \rangle\ \langle \textit{expr} \rangle \\
 & | & \langle \textit{ident} \rangle\ \texttt{(}\ \langle \textit{expr} \rangle^{\star}_{,}\ \texttt{)} \\
 & | & \texttt{[}\ \langle \textit{expr} \rangle^{\star}_{,}\ \texttt{]} \\
 & | & \texttt{(}\ \langle \textit{expr} \rangle\ \texttt{)} \\
\langle \textit{binop} \rangle & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{//} \mid \texttt{\%} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{>} \mid \texttt{<} \mid \texttt{!=} \mid \texttt{==} \\
 & | & \texttt{and} \mid \texttt{or} \\
\langle \textit{const} \rangle & ::= & \langle \textit{integer} \rangle \mid \langle \textit{string} \rangle \mid \texttt{True} \mid \texttt{False} \mid \texttt{None}
\end{array}
$$

Figure 1: Grammar of Mini Python.

2 Static Typing

Though Python is a dynamically-typed language, Mini Python is simple enough to allow us some checks at compile time. These checks are the following:

1. Any function used in an expression must be
   - either a function that was previously defined;
   - either the function we are currently defining (a recursive function);
   - one of the three built-in functions `len`, `list`, and `range`. (In Mini Python, `print` has built-in syntax and thus is not considered as a function.)

   In particular, there are no mutually recursive functions in Mini Python.

2. The names of the functions declared with `def` are pairwise distinct, and distinct from `len`, `list`, and `range`.

3. Function arity must be obeyed, *i.e.*, a function defined with n formal parameters must be called with exactly n actual parameters. Functions `len`, `list`, and `range` all have one parameter.

4. Built-in functions `list` and `range` are exclusively used in the compound expression `list(range(`e`))`.

5. The formal parameters of a function must be pairwise distinct.

6. The scope of variables is statically defined. A variable is either local to a function or global. A local variable x is introduced either as a function parameter or via an assignment $x = e$ anywhere in the function body. The scope of a local variable extends to the full body of the function. A global variable is introduced via an assignment in the toplevel code (the code outside of function definitions at the end of the program). Hint: It is convenient to see the toplevel code as the body of a `main` function. This way, global variables are simply variables that are local to function `main`.

   Note that it is not possible to shadow a variable in Mini Python. We do not (and could not) check at compile time that a variable is defined before being used.

3 Semantics

Any value in Mini Python has a *dynamic type.* This type is assigned to the value at creation time and it cannot be modified thereafter. There are five possible types: `none`, `bool`, `int`, `string`, and `list`. The semantics of an operation can vary according to the type of its actual parameters. In some cases, it can lead to a runtime failure. In this case, the code produced will display a message of the form

```
error: some message
```

and will terminate with exit code 1. The message does not matter but the exit code is important, as it will be used in automated tests.

Values. The `none` type contains a single value, noted as `None`. In particular, this is the value returned by a function that reaches its return point without encountering a `return` statement. The `bool` type is the Boolean type, with two possible values are `False` and `True`. The `int` type is for signed 64-bit integers. Type `string` is for character strings. Finally, type `list` is for lists, which are heterogeneous and possibly empty. The elements of a list can be modified in place, with the statement x`[`e_1`] = `e_2, but the length of a list cannot be modified.

Built-in Function `print`. The `print` statement displays the value passed as a parameter, followed by a newline. The display format is as follows:

type	*printing*
`none`	`None`
`bool`	`False`, `True`
`int`	in decimal, *e.g.* `42`
`list`	`[`e_1`, ..., `e_n`]`, *e.g.* `[1, 2, 3]`
`string`	without quotes, *e.g.* `abc`
	`\n` is printed as a newline
	`\"` is printed as `"`

Boolean Condition. The statement `if` and the operators `and`, `or`, and `not` all accept operands of any type, with the following semantics:

- `None`, `False`, the integer 0, the empty string, and the empty list are interpreted as false;
- any other value is interpreted as true.

Operators `and` and `or` are lazily-evaluated: the second operand is evaluated only if needed, *i.e.* if the value of the first operand does not determine the final value. The result of e_1 `and` e_2 (resp. e_1 `or` e_2) is either the value of e_1 or the value of e_2. For instance, `0 and [1,2]` evaluates to `0` and `1 and [1,2]` evaluates to `[1,2]`.

Iteration. The statement `for x in e: s` first evaluates expression `e`, which must be of type `list`. Then, for each value v in this list, in order, it assigns v to the variable `x` and executes the statement `s`.

Operators. Subtraction (`-`), negation (unary `-`), multiplication (`*`), division (`//`), and modulo (`%`) are only defined on type `int`, with signed 64-bits machine arithmetic[1]. Operator `+` is overloaded, with two parameters of the same type and the following semantics:

type of parameters	semantics
`int`	arithmetic addition
`string`	concatenation (in a new string)
`list`	concatenation (in a new list)
otherwise	failure

[1] Python's division is actually not machine's division, but we accept this difference.

Comparison Operators. The six comparison operators (`<`, `<=`, `>`, `>=`, `==`, `!=`) always return a Boolean value. For any comparison operator, Boolean operands are interpreted as integer values (`False` being 0 and `True` being 1). Operators `==` and `!=` are defined for operands of any type, including two operands of different types. Operators `<`, `<=`, `>`, and `>=` are limited to the following cases:

type of operands	semantics
`bool` or `int`	arithmetic comparison
`string`	lexicographic comparison
`list`	lexicographic comparison

The comparison is structural: when comparing lists, the comparison operation is recursively applied to the list elements.

Built-in Functions. Function `len` is only defined on types `string` and `list`. It returns the length of the string and of the list, respectively. The expression `list(range(`e`))` is defined for an expression e that evaluates to some integer $n \geq 0$. It returns a list of integers $[0, 1, \ldots, n-1]$.

Differences w.r.t Python. There are some small runtime differences between Python and Mini Python, including (but not exhaustively):

	Python	Mini Python
arithmetic	unbounded	signed 64 bits
string display within lists	`['abc']`	`[abc]`
multiplying a string/list and an integer	defined	undefined
access $l[i]$ with $i < 0$	defined	undefined

Your compiler will not be tested on programs for which there is a distinct runtime behavior between Python and Mini Python.

4 Code Generation

The aim is to produce a simple but correct compiler. In particular, we do not attempt to do any kind of register allocation, but simply use the stack to store any intermediate calculations. Of course, it is possible, and even desirable, to use some x86-64 registers locally. Memory is allocated using `malloc` and no attempt will be made to free memory.

Value Representation. Though Python is an object-oriented language, **we are not** compiling Mini Python using objects. We propose a simple compilation scheme (but you are free to use any other) where a value is always a pointer to some heap-allocated block. The first word (64 bits) of this block is an integer tag that encodes the type.

`none`	0	0				
`bool`	1	n	(where n is either 0 or 1)			
`int`	2	n				
`string`	3	n	0-terminated string			
`list`	4	n	v_0	v_1	$\dots$	v_{n-1}

where n is a 64-bit integer and v_i are values (*i.e.*, pointers to other heap-allocated blocks). For strings and lists, the integer n is the length. For a string, the third component is a 0-terminated string stored in $n+1$ bytes (we assume ASCII strings in Mini Python).

Stack Layout. We suggest a compilation scheme where all parameters are passed on the stack (each of them being a 64-bit pointer), and where the return value is in register `%rax`. The stack frame is as follows:

	parameter n
	$\vdots$
	parameter 1
	return address
`%rbp` $\rightarrow$	saved `%rbp`
	local variables
	$\vdots$
	temporary values
`%rsp` $\rightarrow$	$\vdots$
	$\downarrow$

Local variables are allocated on the stack. The top of the stack is used to store intermediate computations, such as the value of e_1 during the evaluation of e_2 in a binary operation $e_1 \oplus e_2$.

With recent versions of the `libc`, it is important to have a 16-byte stack alignment when calling library functions such as `malloc` or `printf` (this is required by the System V Application Binary Interface). Since it is not always easy to ensure stack alignment when calling library functions (because of intermediate computations temporarily stored on the stack), it may be convenient to introduce wrappers around library functions, as follows:

```
my_malloc:
      pushq   %rbp
      movq    %rsp, %rbp
      andq    $-16, %rsp  # 16-byte stack alignment
      call    malloc
      movq    %rbp, %rsp
      popq    %rbp
      ret
```

These wrappers are simply concatenated to the generated assembly code — and of course any call to `malloc` is replaced with a call to `my_malloc`. For instance, following the value representation schema given above, allocation of integers could be done as follows:

```
P_alloc_int:
        pushq   %rbp
        movq    %rsp, %rbp
        pushq   %rdi         # the integer n to allocate is stored in %rdi
                             # and now we push it into the stack;
        andq    $-16, %rsp   # stack alignment;
        movq    $16, %rdi    # how many bytes you want to allocate;
        call    malloc       # the new allocated address is in %rax,
                             # which is a 16 bytes = 2 * 8 = 2 * 64 bits
                             # segment;
        movq    $2, (%rax)   # put the tag of an integer block
                             # in the address pointed by %rax;
        movq    -8(%rbp), %rdi # get back the value of n, which is now on
                                # the stack, at address %rbp - 8 bytes;
        movq    %rdi, 8(%rax)  # put the value of n at address
                                # %rax + 8 bytes;
        #### Now, we have the following, contiguous block allocated:
        ####
        ####        +---------+---------+
        ####        |    2    |    n    |
        ####        +---------+---------+
        ####        | 8 bytes | 8 bytes |
        ####        +---------+---------+
        ####
        movq    %rbp, %rsp
        popq    %rbp
        ret                  # the result is in %rax
```

Here is a list of functions from the C standard library that you may want to use (feel free to use any other):

- `void *malloc(size_t size);`
  `malloc(`n`)` returns a pointer to a freshly heap-allocated block of size n
  You don't have to free memory.

- `int putchar(int c);`
  `putchar(`c`)` writes the character c to standard output (ignore the return value)

- `int printf(const char *format, ...);`
  `printf(`f`,...)` write to standard output according to the format string (ignore the return value). *Register `%rax` must be set to zero before calling `printf`.*

- `int strcmp(const char *s1, const char *s2);`
  compare strings `s1` and `s2`, returning 0 if they are equal, a negative value if `s1` is smaller than `s2`, and a positive value if `s1` is greater

- `char *strcpy(char *dest, const char *src);`
  copy the 0-terminated string `src` to `dest`, including the `'\0'` character
  (ignore the return value)

- `char *strcat(char *dest, const char *src);`
  appends the 0-terminated string `src` at the end of string `dest`, assuming there is enough space (ignore the return value)

Important Notice. Grading involves (for one part only) some automated tests using small Python programs with `print` commands. They are compiled with your compiler, and the output is compared to the expected output. This means you should be careful in compiling calls to `print`.

5 Project Assignment (due June 7, 23:59)

The project must be done **alone or in pair, preferably Java or OCaml**. It must be delivered by email, to address mjp.pereira@fct.unl.pt, as a message with subject `[ICL25-Project]`. Your message should contain a compressed archive containing a directory with your CLIP number(s) (*e.g.* `12345-54321`). Inside this directory, source files of the compiler must be provided (no need to include compiled files). The command `make` must create the compiler, named `minipython`. The compilation may involve any tool (such as `dune` for OCaml) and the `Makefile` can be as simple as a call to such a tool. The command `minipython` may be a script to run the compiler, for instance if the compiler is implemented in Java.

The archive must also contain **a short report** explaining the technical choices and, if any, the issues with the project and the list of whatever is not delivered. The report can be in format ASCII, Markdown, or PDF.

The command line of `minipython` accepts an option (among `--parse-only` and `--type-only`) and exactly one file with extension `.py`. If the file is parsed successfully, the compiler must terminate with code 0 if option `--parse-only` is on the command line. Otherwise, the compiler moves to static type checking. Any type error must be reported as follows:

```
file.py:4:6:
bad arity for function f
```

The location indicates the filename name, the line number, and the column number. Feel free to design your own error messages. The exit code must be 1.

If the file is type-checked successfully, the compiler must exit with code 0 if option `--type-only` is on the command line. Otherwise, the compiler generates x86-64 assembly code in file `file.s` (same name as the input file, but with extension `.s` instead of extension `.py`). The x86-64 file will be compiled and run as follows

```
gcc file.s -o file
./file
```

possibly with option `-no-pie` on the `gcc` command line.

An Extension of Your Choice. You can easily get extra points in this project. To do so, implement an extension of your choice. This can be

- a better test suite;
- cope with runtime errors (explained below);
- the support of another Python construct;
- more static analysis;
- very informative error messages;
- a compiler optimization;
- etc.

This extension must be described in the report and illustrated with test files.

Programs with runtime errors. In the supplied test suite, there are some tests that represent Mini Python programs that only fail at runtime. In other words, examples of Mini Python programs that must pass the type-checking phase of your compiler, but will still produce an error at runtime (just like with regular Python). For instance,

```
print(1 + "foo")
```

At first, you can simply ignore this class of programs and concentrate on well-behaved programs. Producing x86-64 assembly code for a program with runtime error is an extra in this project.

A runtime error must be reported, but no location nor a detailed message is expected so it is fine to simply output

```
error
```

and terminate with exit code 1.

6 For Mx MacOS Machines

The new Apple MacOS machines are powered by an ARM cortex, being known as the Mx family. The assembly these processors run is **not** x86-64. So, you cannot test your solution directly by running `gcc` on your machine.

In order to circumvent this obstacle, I have tested the following solutions:

1. Use the UTM application, `https://mac.getutm.app`, in Emulation mode to run a x86-64 based Linux distribution on your MacOS machine;
2. Use an online x86-64 interpreter, such as `https://www.jdoodle.com/compile-assembler-gcc-online`.

If you come up with any other solution, please do share with me and your colleagues :)