CSCI 334:
Principles of Programming Languages

Lecture 13: Language Architecture

Instructor: Dan Barowy
Williams

1

Topics

How do programs run?
Parser Combinators

2

Your to-dos

1. Lab 6, **due Monday 4/8** (partner lab)
2. Read *Parser Combinators* for **next week**.

3

Announcements

- **WCMA visit**, Thursday, April 11.
- Colloquium: **Algorithmic Approaches to Subset Sum (and Other Hard Problems)** at 2:35pm in Wege Auditorium.

The Subset Sum problem is the most fundamental NP-complete problem concerned with adding numbers together. However, progress on exact algorithms for this problem has been slow: Since Horowitz and Sahni's 1974 invention of the "Meet-in-the-Middle" approach, our best algorithms have relied on simple enumeration and dynamic programming strategies. The lack of an algorithm for Subset Sum that leverages our modern understanding of addition points to important gaps in our knowledge about the behavior of the integers.

4

Quiz

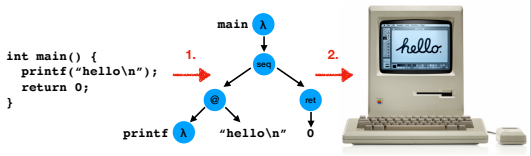
5

See posted solution.

How do programs run?

6

How do programs run?



1. lexical analysis ("front-end")
2. evaluation ("back-end")

7

A programming language is a program, and it has parts. Let's explore those parts. To a first approximation, a programming language has two parts: the front-end and the back-end.

The front-end is responsible for lexical analysis which includes parsing. The output of a lexical analysis is an abstract syntax tree.

The back-end is responsible for either generating a new program in a different form (as in a compiler) or for running the program directly (as in an interpreter). We will focus primarily on interpreters in this class.

Front-end: the parser

A **parser** is a **function** that takes as input a string of symbols conforming to the rules of a formal grammar. If the string is not a valid sentence in the language, the parser **rejects** the string. If the string is a valid sentence in the language, the parser **accepts** the string and outputs a data structure that **represents the meaning of the sentence**.

For programming languages, meaning is generally represented in the form of an **abstract syntax tree** (AST). In an AST, conventionally, interior nodes are operations, and leaves are data.

8

Front-end: the parser

The subject of today's lesson.

9

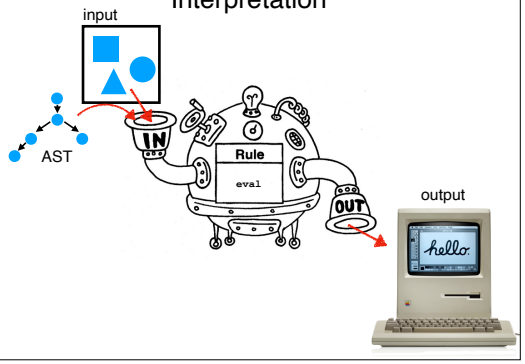
Back-end: the evaluator

There are two kinds of back-end:

1. Interpreter
2. Compiler

10

Interpretation

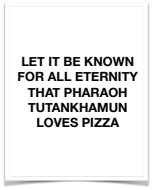


11

An interpreter is a function. We often refer to its role as "evaluation" and so the main function of an interpreter is often called "eval". The interpreter takes an AST and some program input and directly executes it. Remember, an interpreter is itself a program, and it can written in any general-purpose language (C, Java, etc). We will write interpreters in this class using F#.

Interpretation Downsides

- Usually (very) slow
  (often 100-200x slower than compilation)



12

Interpreters are usually pretty slow. The intuition is that of having to translate a document. It's like having to pick through every word and translate it one-by-one. We have to do this procedure even if we've seen the document before. Slow.

Interpretation Advantages

- An interpreter is "just a program" so debugging a language is the same as debugging any other program.

13

Interpreters have the advantage that, compared to compilers, are relatively easy to understand and debug. They're a great starting point if you want to try your hand at designing and implementing a programming language.

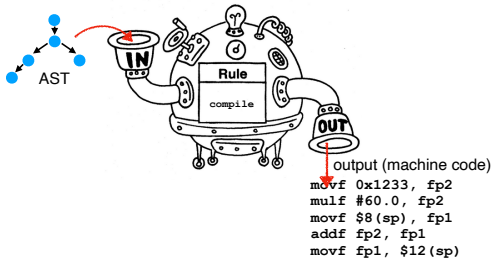
Some interpreted languages

- Shell (e.g., bash)
- Python
- Ruby
- MATLAB
- R
- (sort of) Java and JavaScript

14

You've almost definitely used an interpreted language before.

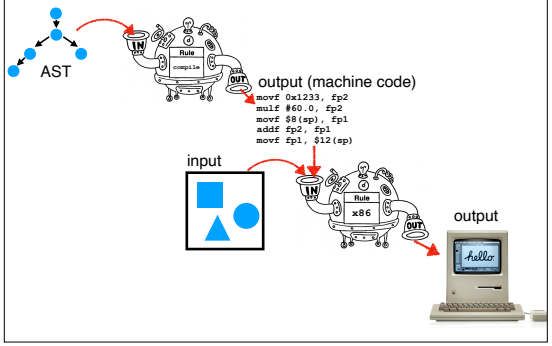
Compilation



15

Compilation is quite a different approach. Instead of running the program directly, a compiler generates a version of the program in a different form. That form is usually machine code, which is the computer's "native" language. Machine code can be executed efficiently on a machine. However, compilers can also translate to a different language. For example, the TypeScript compiler generates JavaScript.

Compilation



16

Note that the compiler is invoked separately from the program. This allows one person (a developer) to run the compiler and another person, like a customer, to run the program. The customer does not need to pay the cost of analyzing the program, which is one reason why compiled programs are faster than interpreters. The compilation cost is paid for by the developer. If you've used commercial software (Windows, macOS, Chrome, etc), you are almost always using a compiled program.

Some compiled languages

- C
- C++
- Go
- FORTRAN
- Java (sort of)
- C# (ditto)
- F# (ditto)

17

There are many compiled programs. These are probably the most popular.

Compilation Advantages

- Usually (very) fast
  (only 1.5-2X slower than hand-optimized assembly code)
- Compiled program is in machine (binary) format; difficult to debug the language itself.

18

Compilation is not necessarily faster than interpretation. The intuition is that compilation simply divides the work differently. Returning to our analogy of having to translate a document in a foreign language, compilation is like translating the document (much like how an interpreter does it), but by saving the translation of to the side. The next time you need the translation, you simply pull it from your pocket instead of re-translation word for word. Much faster.

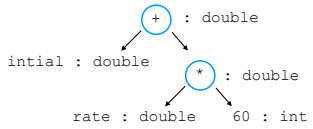
Compilation Example

19

Let us walk through a simple compilation step by step so that you can see how it works. We won't spend much more time discussing compilation in this class, so consider this a brief intro to whet your appetite.

Parsing

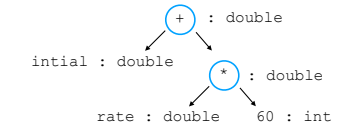
```
double position = initial + (rate * 60)
```



20

Suppose we have a simple one line program in a language like Java or C. We first run a lexical analysis and produce an AST. One such AST might look like this. Observe that our multiplication operation has two operands of different types. This may seem like no big deal to you, but it's generally a no-no for computers. We will need to convert that integer value into a double to be able to continue.

Intemediate Representation




```
temp1 = convert_int_to_double(60)
temp2 = mult(rate, temp1)
temp3 = add(initial, temp2)
position = temp3
```

21

The back-end of the compiler usually starts by generating a program in a new form we call an intermediate representation (IR). The IR is not the final output. But the IR will help guide us to our final output. We obtain the IR by traversing the tree in a depth-first manner. Note that every intermediate result needs to be stored somewhere on a computer, and when generating an IR, the compiler will usually put these intermediate results in abstract memory locations called “temporaries.” If, at the end of the compilation process, any temporaries remain, they must be explicitly assigned to memory locations. As you will see, though, compilers usually try to minimize the use of temporaries when they can.

“Optimization”

```
temp1 = convert_int_to_double(60)
temp2 = mult(rate, temp1)
temp3 = add(initial, temp2)
position = temp3
```

```
temp1 = mult(rate, 60.0)
position = add(initial, temp1)
```

22

If a human were to generate an intermediate representation, they would immediately notice that the machine-generated form is inefficient. For example, there really is no reason to convert the integer 60 into a floating-point 60.0 every single time the program runs. It’s a literal value— why not convert it just once at compile-time? Likewise, why assign the result of the add to a temporary and then assign the temporary to the final variable when we could just assign the add directly to the final variable? A compiler’s optimizer is designed to shorten a program while retaining its

intended function. Note that an optimizer rarely (if ever) generates an "optimally fast" program, so the word "optimization" is a bit of a misnomer. However, the program usually does become faster.

Instruction Selection

```
temp1 = mult(rate, 60.0)
position = add(initial, temp1)
```

```
movf rate, fp2
mulf #60.0, fp2
movf initial, fp1
addf fp2, fp1
movf fp1, position
```

23

Lastly, the compiler convert the IR into the final machine code form. Because the IR is simple, and more closely resembles the semantics of machine code than the source language, converting an IR into machine code is usually pretty straightforward. One complicating factor that can have a big impact on performance is where variables are stored. Computers usually have two forms of memory: register memory, which is extremely fast, and main memory, which is relatively slow. A great deal of complexity in this step comes from trying to keep all of the user's variables in fast register memory. When there are too many variables to fit in registers, some of them may need to "spill over" to main memory. These so called "spills" can slow programs down tremendously.

Compilation Downsides

- Compilation can take a long time

- Cannot modify program without source code.
- Hard to evolve language; compilers are complex.

24

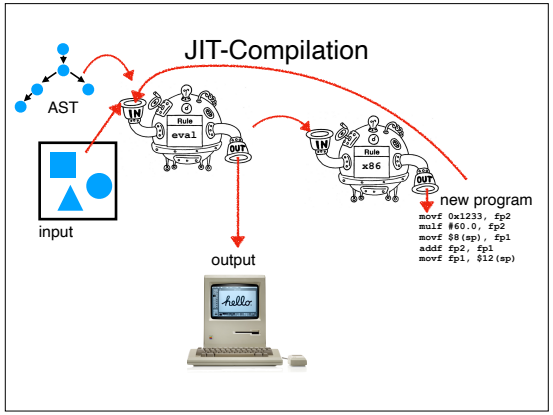
Although compilation can result in faster programs than interpretation, the process itself is complex and slow. Even now, big programs can take hours or days to compile. A typically modern operating system still takes on the order of days to compile.

Some hybrid (JIT) languages

- Java (C#, F#)
- JavaScript

25

There is a third category of programs, however. Just-in-time compiled (JIT) languages. These are some of the most advanced programming languages.



26

A JIT compiler is a hybrid of an interpreter and a compiler. It starts by interpreting the program. For simple programs, that may be all that happens. However, the interpreter keeps track of which functions are executed frequently (the so-called “hot functions”) and those functions are sent off to a compiler running on the side. The compiler generates faster versions of those functions in native machine code which are then patched back into the data structure representation of the program being used by the interpreter. When the interpreter encounters JIT-compiled code, it runs that code natively, usually with a big speedup. JIT compiled programs have the odd property that they tend to get faster the longer they run. We call this phenomenon “warm up.”

History

- Surprisingly, compilers were invented before interpreters.
- More obvious to early engineers.

27

Compilers: History

- Invented by Grace Hopper in 1952 while working on the A-0 and FLOW-MATIC languages.
- Work eventually became the COBOL programming language, still widely in use today.

28

COBOL was one of the first languages that targeted ordinary users, by expressing concepts in an English-like way. Although modern languages like Python and Java may not seem very English-like to you, in comparison to languages before COBOL, they are quite friendly. Modern languages were strongly influenced by Grace Hopper's work.

Compilers: History

I used to be a mathematics professor. At that time I found there were a certain number of students who could not learn mathematics. I then was charged with the job of making it easy for businessmen to use our computers. I found it was not a question of whether they could learn mathematics or not, but whether they would. […] They said, 'Throw those symbols out — I do not know what they mean, I have not time to learn symbols.' I suggest a reply to those who would like data processing people to use mathematical symbols that they make them first attempt to teach those symbols to vice-presidents or a colonel or admiral. I assure you that I tried it. — Grace Hopper

29

Grace Hopper explaining why plan English style languages were important.

Interpreters: History

- Invented by John McCarthy in 1958 while working on LISP.
- Invented as a byproduct of McCarthy's thinking about computation from first principles.
- McCarthy wanted to build computers that could *think*!

- LISP was too resource hungry for most uses at the time.

30

Interpreters were invented shortly after compilers. The first was LISP, which was invented by John McCarthy. McCarthy wanted the programmers in his laboratory to be able to focus on solving problems in AI, not in wrangling with peculiar low-level details of machine code. He wanted the most powerful language he could get, so he patterned it after the lambda calculus. LISP's capabilities were so far ahead of their time that most people had trouble wrapping their heads around it. In this class we are learning a language that was directly influenced by LISP, so by now, you know most of those concepts. Unfortunately, LISP required very expensive computers to run with any kind of acceptable speed which put LISP out of reach of most users for a very long time.

Parsers

31

Let's start looking at the parts of languages in detail now, starting with the front-end task of parsing.

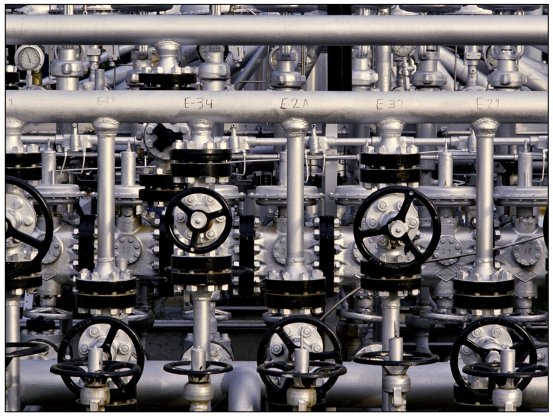
Parser Combinators

32

There are many algorithms for parsing. You could spend an entire semester on just parsing algorithms. They're interesting and still an area of active research. Nevertheless, in the interest of teaching you an approach that you can use quickly, I have settled on parser combinators, which is an approach that provides a straightforward way to build parsers.

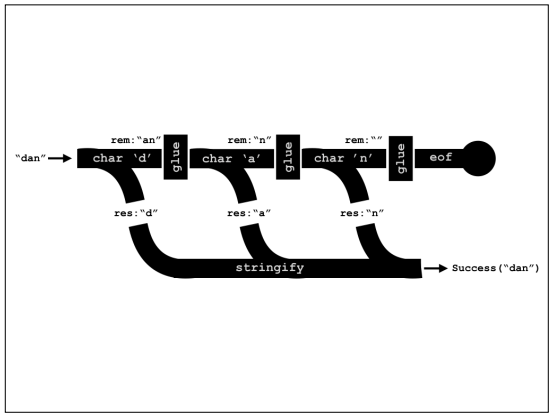
33

I want you to keep an analogy in mind as you use parser combinators: pipes. If you think about pipes, there are two phases of interaction with them: first we build them and then we use them. Parser combinator work much the same way. First we construct a data pipeline, then we put things into the data pipeline. More concretely, this means that when you call a parser combinator function, you are not running the parser; rather you are constructing a parser program. You run it once you are done constructing it.



34

Parser combinators are simple and logical. As long as you are follow the guidelines as you build parsers, you can build sophisticated parsers that are not confusing, much like how engineers can build sophisticated pipelines.



35

Here's a simple parser that only accepts the word "dan." It is made out of three kinds of simple parsers:
1) a character parser (e.g., char 'd' just parses the letter d)
2) an end-of-file parser (eof) that makes sure all of the input has been parsed, and
3) a glue parser that glues two parsers together (e.g., char 'd' and char 'a').

We also see a user-defined function here, stringify, that takes the outputs of the parsers and concatenates them together. When the input is "dan", the parser succeeds. The parser will fail on other inputs.

Parser Combinators

- A kind of recursive decent parser.
- A **recursive descent parser** is a parser built from a set of **mutually recursive procedures** where each such procedure usually **implements one of the productions** of the grammar.
- Recursive descent parsers are "**top-down**," meaning that they recognize sentences by expanding nonterminals, starting from the start symbol.
- "**Bottom-up**" parsers start with *terminal* symbols and work in the opposite direction, often utilizing dynamic programming… these are more common in practice!

36

Basic Primitives

- Input

```
type Input = string * int * bool
```

- Output

```
type Outcome<'a> =
| Success of result: 'a * remaining: Input
| Failure of fail_pos: int * rule: String
```

37

We will use a parser combinator library written by me in this class. It has two basic primitives.

First, we have Input. An Input is a 3-tuple of an input string, our current position in the string, and a flag that says whether the parsers should output debug information.

Second, we have Output. An Output is either Success, which is a 2-tuple of some kind of user-defined result and the remaining unparsed input, or a Failure, which is a 2-tuple of the location of the failure in the input string and a reason why the failure happened.

Basic Primitives

- A parser is

```
type Parser<'a> = Input -> Outcome<'a>
```

- Keep in mind: a parser *is a function*.

38

So a parser is a function that takes an Input and returns an Output.

We will see some example parsers in the next class.

Recap & Next Class

Today:

- Language architecture
- Parser combinators

Next class:

- Growing a Language
- More parsing