

CSCI 334:
Principles of Programming Languages

Lecture 14: Parsing

Instructor: Dan Barowy
Williams

Topics

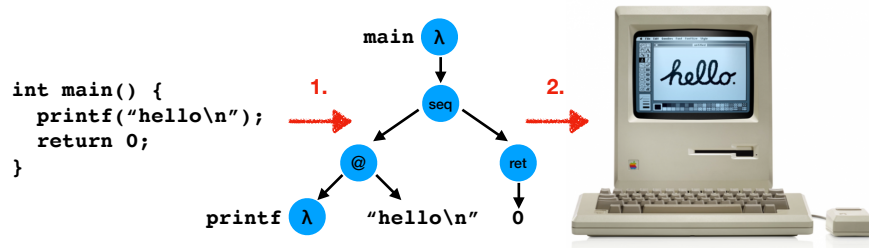
Parts of a language
Parser combinators

Your to-dos

1. Quiz, **due Wednesday 11/8.**
2. Lab 8, **due Sunday 11/13** (partner lab)

How do programs run?

How do programs run?



1. lexical analysis (“front-end”)

2. evaluation (“back-end”)

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.

Front-end: the parser

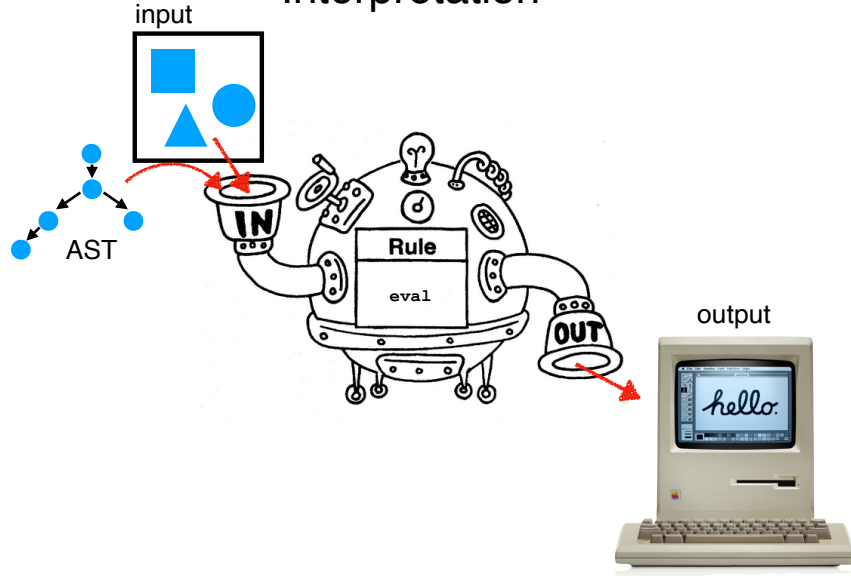
The subject of today’s lesson.

Back-end: the evaluator

There are two kinds of back-end:

1. Interpreter
2. Compiler

Interpretation



Interpretation Downsides

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



LET IT BE KNOWN
FOR ALL ETERNITY
THAT PHARAOH
TUTANKHAMUN
LOVES PIZZA

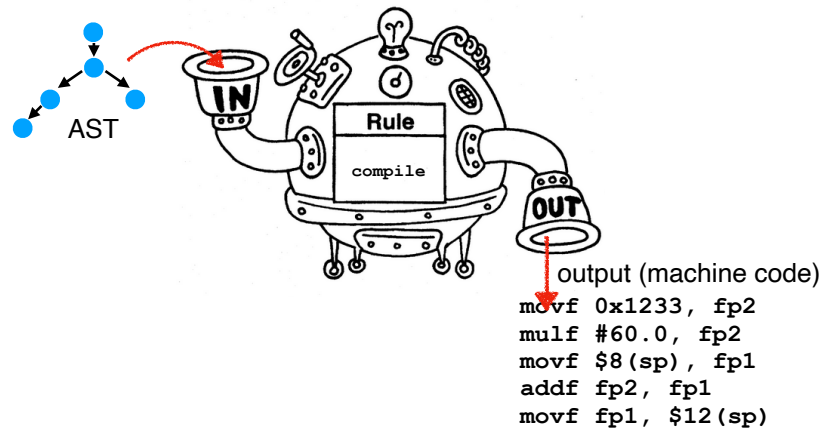
Interpretation Advantages

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

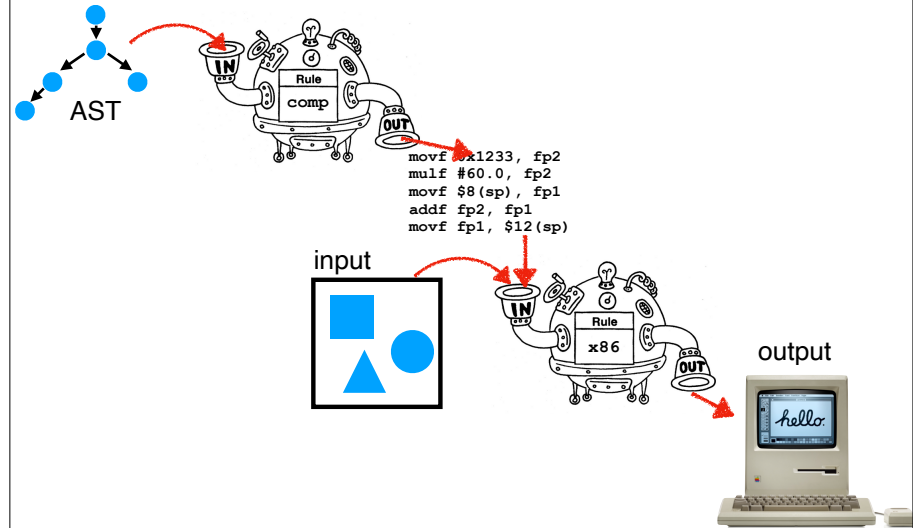
Some interpreted languages

- Most Lisps
- Python
- Ruby
- MATLAB
- R
- (sort of) Java and JavaScript

Compilation



Compilation



Some compiled languages

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

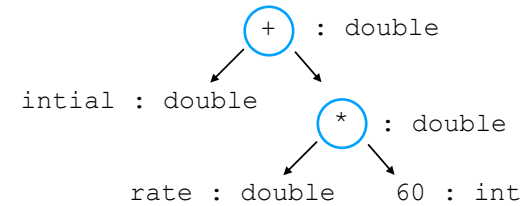
Compilation Advantages

- Usually (very) fast
(often 1.5-2X slower than hand-optimized assembly code)
- Compiled program is in machine (binary) format; difficult to debug a buggy language because many steps separate source program from final output.

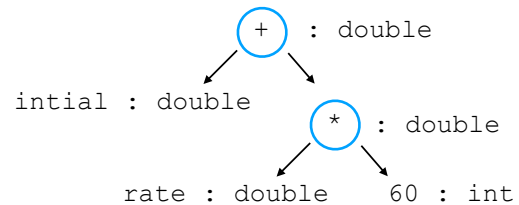
Compilation Example

Parsing

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



Intermediate Representation



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

“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)
```

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
```

Compilation Downsides

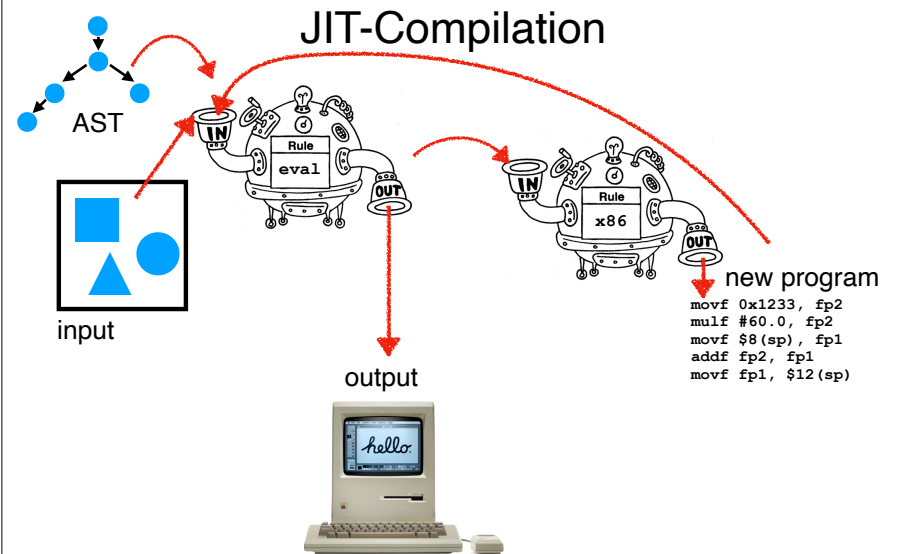
- Compilation can take a long time



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

Some hybrid (JIT) languages

- Java (C#, F#)
- JavaScript



History

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

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.

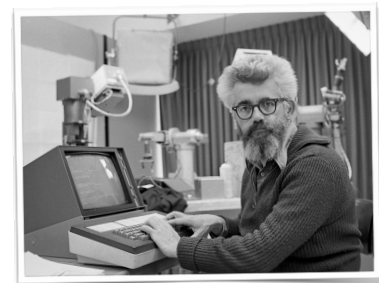


Compilers: History

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

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*!

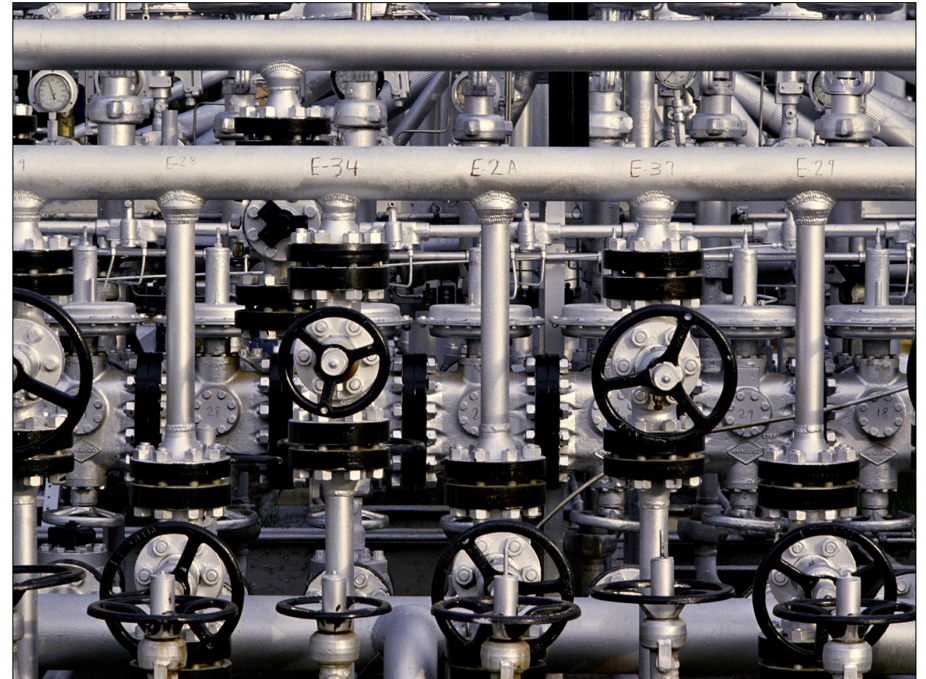


- LISP was too resource hungry for most uses at the time.

Parsers

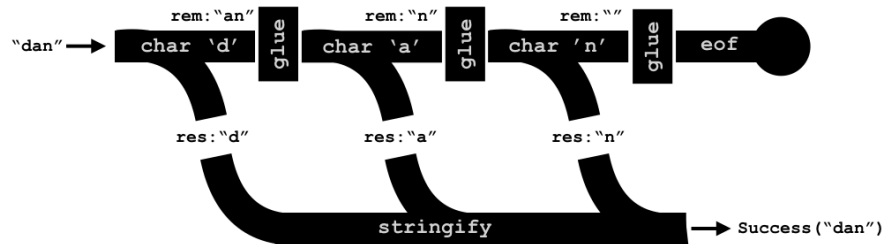


Parser Combinators



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!



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
```

Basic Primitives

- A parser is

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

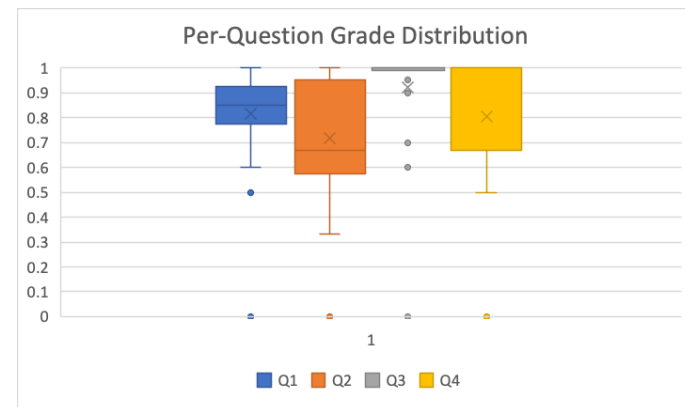
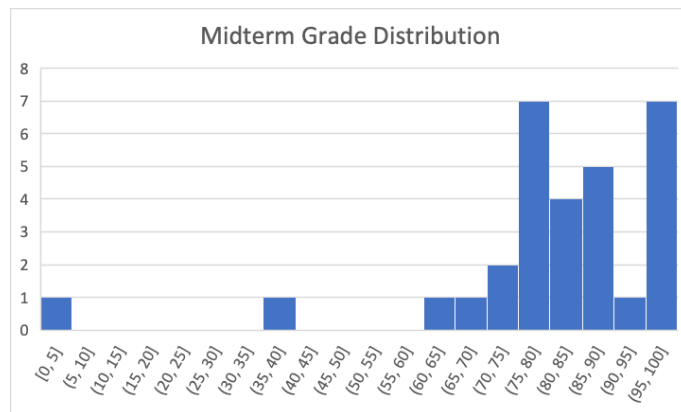
- Keep in mind: a parser *is a function*.

Two varieties of parser

- Parsers that **consume input**. Correspond with grammar terminals.
- Parsers that **combine parsers**. Correspond with grammar non-terminals. Also called “combining forms.”
- For flexibility, you can also have **parsers that do both**.

A very simple terminal parser

- To parse a given char
`pchar(c: char) : Parser<char>`
- Notice that the **generic type** inside <brackets> is the **return type of the parser**.
- So `pchar` returns a *parser*.
- When it is run with an *input*, it returns an `Outcome<char>`.



Recap & Next Class

Today:

- Part of a language
- Parser combinators

Next class:

- Writing a parser