Week 5
Parsing

Hassam

Outline

Compilers

Parsing

Section 1

Compilers

C to executable?

How do you go from "code" to something the computer understands?

C to executable?

How do you go from "code" to something the computer understands?

- What does the computer even understand?

C to executable?

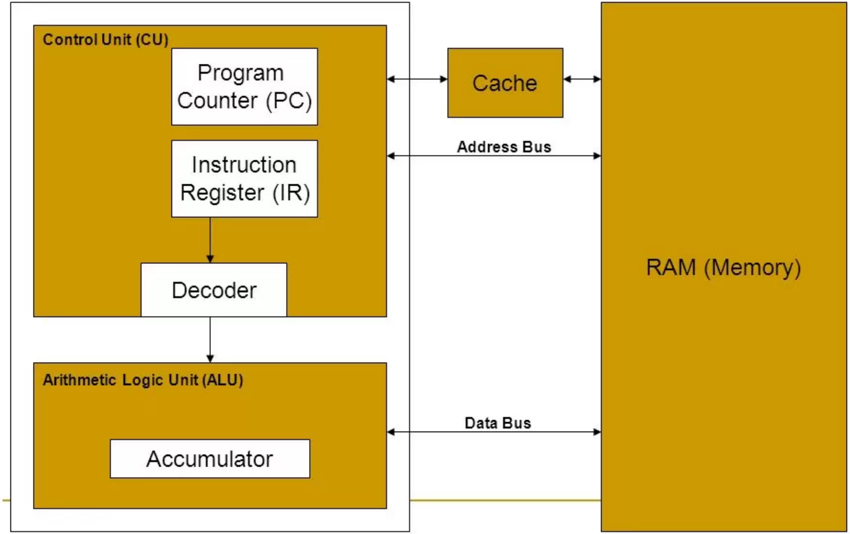
How do you go from "code" to something the computer understands?

- What does the computer even understand?
- What is code?

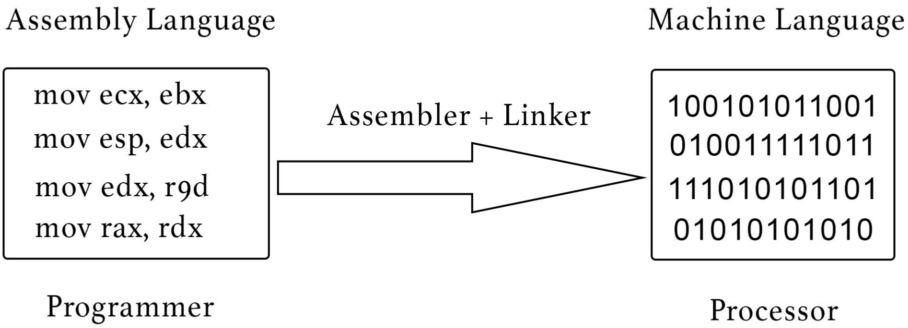
Assembly

@nebu addressed this two weeks ago, but let's review.

CPU Diagram



Control Unit (CU)
Program Counter (PC)
Instruction Register (IR)
Decoder
Arithmetic Logic Unit (ALU)
Accumulator
Cache
Address Bus
Data Bus
RAM (Memory)


Assembly Language
Machine Language
mov ecx, ebx
mov esp, edx
mov edx, r9d
mov rax, rdx
Assembler + Linker
100101011001
010011111011
111010101101
01010101010
Programmer
Processor

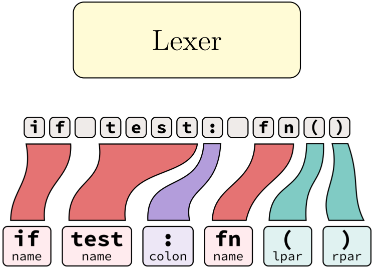
Code?

```c
int main(int argc, char** argv) {
    puts("Hello, World!");
    return 0;
}
```

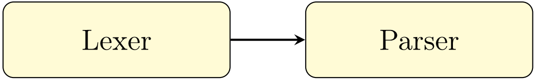
Code?

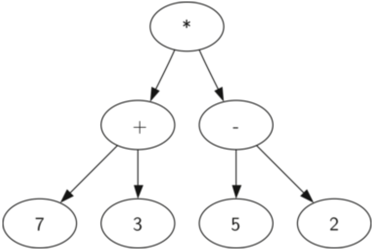
```python
def main():
    print("Hello, world!")
```

What are the steps in between?

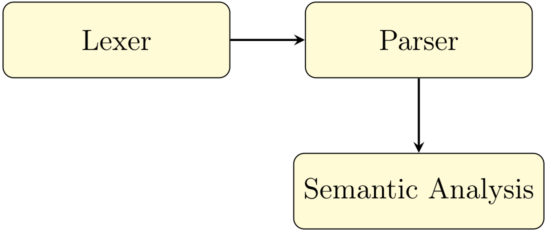


What are the steps in between?



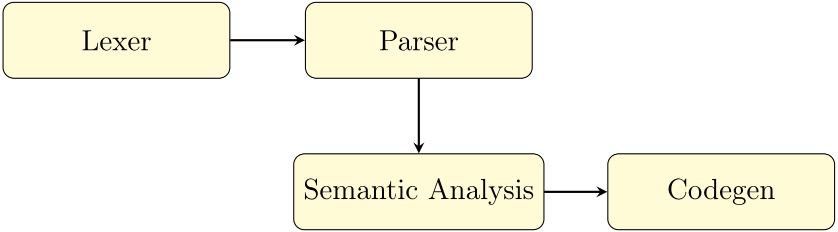


What are the steps in between?



Syntax Errors
Type Checking
Lifetime Analysis

What are the steps in between?



Assembly!
Optimizations

Questions?

Section 2

Parsing

Remember CFGs?

- So far, we have been using CFGs without any reason.
- Soon, we'll use them to prove properties of computation!

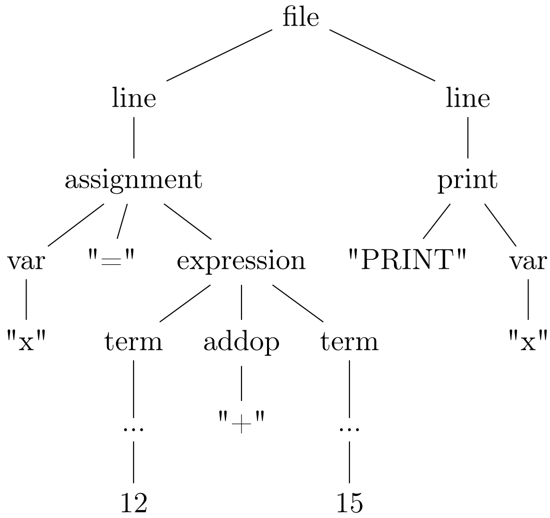
Remember CFGs?

- So far, we have been using CFGs without any reason.
- Soon, we'll use them to prove properties of computation!
- But, sometimes you just want to specify a grammar and parse it, CFGs are useful here too!
- CFGs are ugly though, and we're leaving the world of theory, so we have something better.

Extended Backus–Naur form

```
file = line { line } <EOF>.
line = [ assignment | print | reset ] <NL>.
assignment = var ":=" expression.
print = "PRINT" var.
reset = "RESET".
expression = term { addop term }.
term = factor { mulop factor }.
factor = "(" expression ")" | var | number.
addop = "+" | "-".
mulop = "*".
var = letter { letter | digit }.
number = [ "-" ] digit { digit }.
letter = ('A'-'Z') | ('a'-'z').
digit = ('0'-'9').
```

Parse Tree


file
line
line
assignment
print
var
"="
expression
"PRINT"
var
"x"
term
addop
term
"x"
...
"+"
...
12
15

Parse Tree

```
x = 12 + 15
PRINT x
```

So, PDAs?

- This "language" is representable by a CFG, so should we use a PDA to parse it?

So, PDAs?

- This "language" is representable by a CFG, so should we use a PDA to parse it?
- Let’s not.

LL Parsing

- Reading **L**eft to right, performing a **L**eftmost processing.
- Formally, this type of parser is equivalent to a deterministic pushdown automata. But, much easier to write.
- Requires our grammar to follow some rules, but all grammars (parseable by DPDAs) can be (annoyingly) converted to do so.

Easy to write?

So easy to write:

```
start = F | '(' start '+' F ')'
F = 'a'
```

This parses expressions of the form: $(a + a)$, $((a + a) + a)$, etc.

How do we code this?

```
parseF 'a':xs = F, xs
parseF [] = error

parseS '(':xs = {
    S1, '+':restS = parseS(xs)
    F, ')':rest = parseF(restS)
    return (S (S1, F)), rest
}
parseS x = {
    F, rest = parseF(x)
    return S(F), rest
}
parseS [] = error
```

```
start = F
  | '(' start '+' F ')'
F = 'a'
```

What goes wrong?

- Our code from before is valid and generalizable, but it makes some assumptions.

What goes wrong?

- Our code from before is valid and generalizable, but it makes some assumptions.
- It assumes that there is no "left recursion"

What goes wrong?

- Our code from before is valid and generalizable, but it makes some assumptions.
- It assumes that there is no "left recursion"
- It assumes that there are no common prefixes

What goes wrong?

- Our code from before is valid and generalizable, but it makes some assumptions.
- It assumes that there is no "left recursion"
- It assumes that there are no common prefixes
- It assumes that you can look one character ahead and determine what to do next. This is formally known as LL(1).

What goes wrong?

- Our code from before is valid and generalizable, but it makes some assumptions.
- It assumes that there is no "left recursion"
- It assumes that there are no common prefixes
- It assumes that you can look one character ahead and determine what to do next. This is formally known as LL(1).
- It assumes there's no ambiguity.

What goes wrong?

- Our code from before is valid and generalizable, but it makes some assumptions.
- It assumes that there is no "left recursion"
- It assumes that there are no common prefixes
- It assumes that you can look one character ahead and determine what to do next. This is formally known as LL(1).
- It assumes there's no ambiguity.

Which of these can we fix?

Ambiguity?

```
void func() {
    a < b , c > d;
}
```

Ambiguity?

```
void func() {
    a < b , c > d;
}
```

Is this a template instantiation? Or are we using `operator<` and `operator>` on two variables?

Ambiguity?

```
void func() {
    a < b , c > d;
}
```

Is this a template instantiation? Or are we using `operator<` and `operator>` on two variables?

It's both.

Fixing bad grammars

It turns out, we can fix them all, except for the last one. We need a more powerful tool for ambiguity.

- It is possible to rewrite anything left-recursive to be non-left-recursive.
- Removing common prefixes is called "left factoring".
- We can change from an LL(1) parser to an LL(k) parser, and "look ahead" k steps. This becomes messier and messier the further we look ahead though.

Nearly every mainstream programming language is LL(1). They are convenient to parse, and even more convenient to write.

Fixing left recursion

```
start = start '+' | 'i'
```

This parses i+++++, and onwards. How do we get rid of the left recursion?

Fixing left recursion

```
start = start '+' | 'i'
```

This parses i+++++, and onwards. How do we get rid of the left recursion?

The general strategy is:

- `S = 'x'` becomes `S = 'x'S_new`
- `S = S'x'` becomes `S_new = 'x'S_new`
- Add `S_new =` ϵ

Fixing left recursion

```
start = start '+' | 'i'
```

This parses i+++++, and onwards. How do we get rid of the left recursion?

The general strategy is:

- `S = 'x'` becomes `S = 'x'S_new`
- `S = S'x'` becomes `S_new = 'x'S_new`
- Add `S_new =` ϵ

```
start = 'i' start_new
start_new = '+'start_new | ϵ
```

Fixing common prefixes

```
start = 'a' B | 'a' 'd'
B = 'c'
```

Fixing common prefixes

```
start = 'a' B | 'a' 'd'
B = 'c'
```

Becomes:

```
start = 'a' D
D = B | 'd'
B = 'c'
```

Solving Ambiguity

Remember our CFG from last week?

$$\begin{aligned} S \;\rightarrow\; \varepsilon &\mid S\texttt{ab} \mid \texttt{a}S\texttt{b} \mid \texttt{a}S\texttt{b}S \\ &\mid S\texttt{ba} \mid \texttt{b}S\texttt{a} \mid \texttt{b}S\texttt{a}S \end{aligned}$$

We can try to fix the left recursion and the common prefixes, but we'll get stuck in an infinite loop. This grammar is "ambiguous", we would need to look ahead infinitely many steps to parse it. We need non-determinism to parse this.

Backtracking to the rescue?

```
def parseS(inp):
    if inp == '': return S(), ''
    try:
        s, rest = parseS(inp)
        if rest[0] == 'a':
            assert rest[1] == 'b'
            return S(s, 'a', 'b'), rest[2:]
        elif rest[0] == 'b':
            assert rest[1] == 'a'
            return S(s, 'b', 'a'), rest[2:]
        else:
            assert False
    except Exception:
        assert inp[0] == 'a'
        try:
            s, rest = parseS(inp)
            ...
```

Yikes

This is messy, and it's slow! Try finishing it by yourself.
Most modern compilers for non-LL(1) languages [C and C++ :( ] use handwritten backtracking parsers like this one.

Writing parsers for fun and profit

Writing parsers is actually very fun! I have had four interviews this semester that required writing a custom parser.

- Try writing parsers for some of the simple examples we discussed today in your favorite language.
- Go meta. Write a "parser compiler" that takes some basic version of EBNF and turns it into code that runs a parser.
- So much more to explore. LR(k), LALR(k), etc.
- Parsing is "solved", more interesting problems in compilers now.

Goodbye

Any sufficiently complicated C or Fortran program contains an ad hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp.

— Greenspun's tenth rule of programming (1993)