Compilers

Compilers

- Ras Bodik (Google's Deepmind, UW professor):

  Don't be a boilerplate programmer. Instead, build tools for users and other programmers. Take historical note of textile and steel industries: do you want to build machines and tools, or do you want to operate those machines?

- Steve Yegge (Famous Blogger, at Sourcegraph):

  If you don't take compilers then you run the risk of forever being on the programmer B-list: the kind of eager young architect who becomes a saturnine old architect who spends a career building large systems and being damned proud of it.

What does a Compiler Do?

- A **compiler** is software that translates computer code written in one programming language (the **source** language) into another language (the **target** language).
- Typescript translates a typed superset of Javascript into pure Javascript.
- A C Compiler translates from a high-level language to assembly.
- Babel is a Javascript to Javascript compiler that allows you to use modern features in older browsers by polyfilling them for you.
- LLVM (The compiler backend for Clang) supports many languages (C, C++, Objective-C, Swift, Rust, Go, Fortran, D, Haskell, many more) and emits optimized assembly after lowering to an IR.

How does it do it?

- Tokenization
- Parsing
- Semantic Analysis (Optional)

- Lower to an IR (Optional)
- Optimization (Optional)
- Codegen

Tokenization

Tokenization involves turning a textual program into strings with a set meaning.

```
int main(void) { return 5; }
```

```
[Keyword("int"), Identifier("main"), LParen, Keyword("void"),
RParen, LBrace, Keyword("return"), Int(5), Semicolon, RBrace]
```

You'll want to keep spans for error handling, annotating each token with its location in the source file in case of errors.

Parsing

Parsing turns these tokens into an Abstract Syntax Tree (AST) which is a form of the program that can be traversed in order to generate output.

Say we have a language of this form, which supports addition, subtraction, multiplication, division, parentheses, and order of operations:

```
> 1 + 2 # 3
> 2 - 1 # 1
> 3 * 4 # 12
> 12 / 2 # 6
> (1 + 2) * 4 # 12
> 1 + 2 * 4 # 9
```

Parsing Continued

We would have a grammar like this:

```
Expr   -> Term { ("+"|"-") Term }
Term   -> Factor { ("*"|"/") Factor }
Factor -> Number | "(" Expr ")"
```

One approach is recursive descent which is a natural translation of this grammar into code.

Recursive Descent in Rust

```rust
pub fn parse_expr(&mut self) -> Result<f64, String> {
    let mut value = self.parse_term()?;

    while let Token::Plus | Token::Minus = self.current_token {
        let op = self.current_token.clone();
        self.consume(op.clone())?;
        let right = self.parse_term()?;
        value = match op {
            Token::Plus => value + right,
            Token::Minus => value - right,
            _ => unreachable!(),
        };
    }
    Ok(value)
}
```

```rust
fn parse_term(&mut self) -> Result<f64, String> {
    let mut value = self.parse_factor()?;

    while let Token::Star | Token::Slash = self.current_token {
        let op = self.current_token.clone();
        self.consume(op.clone())?;
        let right = self.parse_factor()?;
        value = match op {
            Token::Star => value * right,
            Token::Slash => value / right,
            _ => unreachable!(),
        };
    }
    Ok(value)
}
```

```rust
fn parse_factor(&mut self) -> Result<f64, String> {
    match &self.current_token {
        Token::Number(n) => {
            self.consume(self.current_token.clone())?;
            Ok(*n)
        }
        Token::LParen => {
            self.consume(Token::LParen)?;
            let expr_val = self.parse_expr()?;
            self.consume(Token::RParen)?;
```

```
            Ok(expr_val)
        }
        _ => Err(format!("Unexpected token: {:?}", self.current_token)),
    }
}
```

Semantic Analysis (Optional)

Let's say we have a language that has strings and numbers, where you can add numbers or concat strings:

```
> "hello " + "world" # "hello world"
> 10 + 20 # 30
> "hello " + 30 # Some kind of error
```

We want to check for the last case and throw an error here:

```
fn validate_binop(bin_op: BinOp) -> Result<BinOp, Error> {
    let BinOp { op, l, r } = BinOp;
    if op == Op::Add {
        if (l.is_string() && r.is_number()) || (l.is_number() &&
        r.is_string()) { return Err(Error::InvalidBinOp); }
    }
    Ok(bin_op)
}
```

Lower to IR

We may want to turn our AST into a form that's more amenable to optimization. One form is Three Address Codes (TAC). LLVM uses Single Static Assignment (SSA) but we'll stick to TAC.

```
int a = 1; int b = 2; int c = 3; int d = 4;
a = b + c + d;
b = a * a + b * b;
```

This can be turned into a TAC like so: (each operation has a target, and at most two operands):

```
a = 1
b = 2
c = 3
d = 4
_t0 = b + c;
_t1 = _t0 + d;
a = _t1;
```

```
_t2 = a * a;
_t3 = b * b;
b = _t1 + _t2;
```

Lower to IR (continued)

We can then optimize this program:

There is one unnecessary move:

```
_t0 = b + c;
_t1 = _t0 + d;
a = _t1;
```

```
_t0 = b + c;
a = _t0 + d;
```

Lower to IR (continued)

We can do some peephole optimizations over and over again till we get this result. Note that since we don't use a or b, technically we can delete the program itself.

```
a = 9
b = 5
```

One interesting optimization

One (there are many interesting optimizations, take a look at Hacker's Delight for more) optimization is called Strength Reduction: take a statement and replace it with something that's cheaper to calculate but gives you the same result.

One example:

```
n * 2 // this can take multiple cycles, but your computer + compiler
      // optimize it
```

```
n << 1 // this takes one cycle always
```

Accessing arrays in a loop

An example of looping through an array to print (1, 2, 3) on their own lines:

```
int array[3] = {1, 2, 3};
```

```
for (int i = 0; i < 3; i++) {
```

```
    printf("%d\n", array[i]);
}
```

Which lowered looks like this. Note there's a multiply to iterate the array:

```
array_ptr = $0800 # some random address
size_of_item = 4 # ints are 4 bytes
load i = 0
for the next three times:
    tmp = load array[i]
    print tmp
    i += 1
    array_ptr += size_of_item * i
```

Accessing arrays in a loop (continued)

We said the multiply is expensive. Let's replace it with addition:

```
array_ptr = $0800 # some random address
size_of_item = 4 # ints are 4 bytes
load i = 0
offset = 0
for the next three times:
    tmp = load array[i]
    print tmp
    i += 1
    offset += size_of_item
    array_ptr += offset
```

Codegen

Finally, you lower the IR or AST into its target language. This is a pretty straightforward translation, keeping in mind that some things may not be supported in the target language, so those require extra handling.

Extra: Testing

How do you test a compiler?

There are infinitely many programs for any turing complete language, thus the set of programs a compiler can accept is infinitely many. Writing tests one by one (a whole program) would take you infinite time.

One way is by fuzzing: generating random programs that fit the semantics of the input language and making sure they're correctly handled in the output

language. You can also provide incorrect programs and make sure they error out correctly.

Questions?