Compilation

Reminder: Why do we want compilation?

- We want to write:

```
{with {x 3}
      {with {y 4}
            {+ x y}}}
```

- We want to interpret:

```
{{fun {x}
      {{fun {y}
            {+ x y}}
       4}}
 3}
```

- **Solution:** a compiler to translate between the two!

Reminder: What is a compiler?

An ***interpreter*** takes a program and produces a result

A ***compiler*** takes a program and produces a program

- The latter is what we want to bridge the gap between programs we want to **write**
    - and programs we want to **run**

Reminder: What is a compiler?

An ***interpreter*** takes a program and produces a result

A ***compiler*** takes a program and produces a program

- The latter is what we want to bridge the gap between programs we want to **write**
  - and programs we want to **run**

- Note that you can have **both** an interpreter and a compiler for a language
  - Or either, or neither, or many of each!
  - There is no such thing as an "interpreted language" or a "compiled" language
  - And don’t get me started on the word "transpiler"...

Why the gap?

- Writing in a large language, with (technically redundant) conveniences (e.g., **`with`**) is nice
    - Writing an interpreter for such a language, not so much
- Our available interpreter (e.g., CPU) may only support a very restricted language (e.g., machine code)
    - Writing programs in that language may not be productive
- Running a highly-optimized program is nice
    - Writing (and debugging!) that program can be painful

In all these cases, a compiler can bridge the gap

So, we're going to write a compiler to bring **`with`** back

Compiler Basics

A compiler relates three languages

- A source language
  - The language of the **inputs** to the compiler
  - Akin to an interpreter's object language
- A target language
  - The language of the **outputs** of the compiler
- A meta language (or implementation language)
  - The language the compiler itself is written in
  - Same as the meta language of an interpreter

In contrast, an interpreter relates two languages: source and object

Compiler Basics

Examples of language triples (input, output, meta):

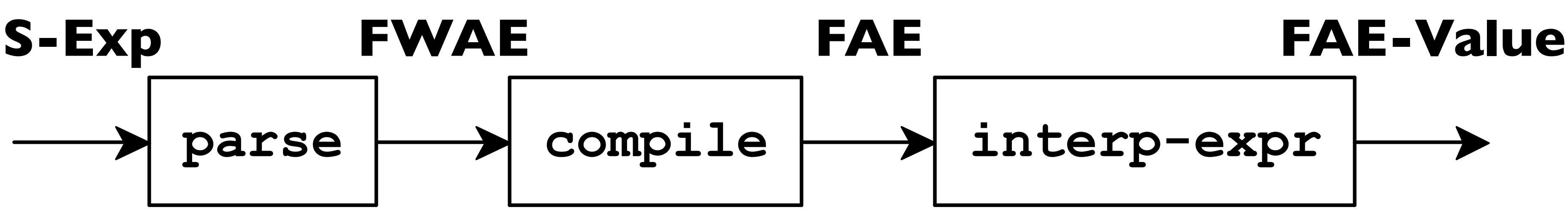
- **GCC:** C, x86-64 machine code, C
- **TypeScript:** TypeScript, JavaScript, TypeScript
- **javac:** Java, JVM bytecode, Java
    - **JVM:** JVM Bytecode, x86-64 machine code, C++ (JIT compiler, so also an interpreter!)
- **Emscripten:** C++, JavaScript, C
    - From a low-level language to a high-level one?
    - Unusual, but still a compiler

Compiler Basics

- The compiler we will write today relates:
  - **FWAE** as the source language
  - **FAE** as the target language
  - **PLAI** as the meta language
- In this case, source and target languages are very close
  - We're using a cannon to kill a fly
    - Overkill, but we get to play with cannons!
  - Take 322 to build a compiler that spans a larger gap

Compiler Basics

- The compiler we will write today relates:
  - **FWAE** as the source language
  - **FAE** as the target language
  - **PLAI** as the meta language
- In this case, source and target languages are very close
  - We're using a cannon to kill a fly
    - Overkill, but we get to play with cannons!
  - Take 322 to build a compiler that spans a larger gap
- Overall system:



FWAE vs FAE

```
<FWAE>  ::=  <num>
          |   {+ <FWAE> <FWAE>}
          |   {- <FWAE> <FWAE>}
          |   {with {<id> <FWAE>} <FWAE>}    ⬅
          |   <id>
          |   {fun {<id>} <FWAE>}
          |   {<FWAE> <FWAE>}

<FAE>  ::=  <num>
         |   {+ <FAE> <FAE>}
         |   {- <FAE> <FAE>}
         |   <id>
         |   {fun {<id>} <FAE>}
         |   {<FAE> <FAE>}
```

FWAE vs FAE

```
(define-type FWAE
  [W-num (n number?)]
  [W-add (lhs FWAE?)
         (rhs FWAE?)]
  [W-sub (lhs FWAE?)
         (rhs FWAE?)]
  [W-with (name symbol?)
          (named-expr FWAE?)
          (body FWAE?)]
  [W-id (name symbol?)]
  [W-fun (param symbol?)
         (body FWAE?)]
  [W-app (fun-expr FWAE?)
         (arg-expr FWAE?)])
; ugh, name clashes...
```

```
(define-type FAE
  [num (n number?)]
  [add (lhs FAE?)
       (rhs FAE?)]
  [sub (lhs FAE?)
       (rhs FAE?)]
  [id (name symbol?)]
  [fun (param symbol?)
       (body FAE?)]
  [app (fun-expr FAE?)
       (arg-expr FAE?)])
```

Compiling FWAE

```
(test (compile (parse `{+ 1 2}))
      (parse-fae `{+ 1 2}))

(test (compile (parse `{with {x 3} x}))
      (parse-fae `{{fun {x} x} 3}))

(test (compile (parse `{+ 2
                           {with {y 7}
                                 {+ y 3}}}))
      (parse-fae `{+ 2
                      {{fun {y} {+ y 3}}
                       7}}))
```

Compiling FWAE

```
; compile : FWAE? -> FAE?
(define (compile an-fwae)
  (type-case FWAE an-fwae
    [W-num (n) (num n)]
    [W-id (name) (id name)]
    ...))
```

Those just translate as is

Compiling FWAE

```
; compile : FWAE? -> FAE?
(define (compile an-fwae)
  (type-case FWAE an-fwae
    ...
    [W-add (l r) (add (compile l) (compile r))]
    [W-sub (l r) (sub (compile l) (compile r))]
    [W-fun (param body) (fun param (compile body))]
    [W-app (fun arg) (app (compile fun)
                          (compile arg))]
    ...))
```

Structural recursion, in case there's a **`with`** somewhere in there

Compiling FWAE

```
; compile : FWAE? -> FAE?
(define (compile an-fwae)
  (type-case FWAE an-fwae
    ...
    [W-with (name bound-expr body)
            (app (fun name
                      (compile body))
                 (compile bound-expr))]))
```

And that's it. The one interesting case.

Optimizing FWAE

- Ok, cool, but now that we have a compiler
  - Can we do more?
- Sure! Let's do a (tiny) bit of optimization

Constant Folding

- Very basic optimization
- 2 + 2 = 4
  - Always true, regardless of the rest of the program
  - (Caveats with machine integers apply)

Constant Folding

- Very basic optimization
- 2 + 2 = 4
  - Always true, regardless of the rest of the program
  - (Caveats with machine integers apply)
- The optimization: `{+ 2 2}` ⇒ `4`
  - For all constant values of 2 and 4

Constant Folding

- Very basic optimization
- 2 + 2 = 4
  - Always true, regardless of the rest of the program
  - (Caveats with machine integers apply)
- The optimization: `{+ 2 2}` ⇒ `4`
  - For all constant values of 2 and 4
- But I never write code like that!
  - Compilers do, though
  - Often used to "clean up" after other optimizations

Constant Folding

```
(test (compile (parse `{+ 1 2}))
      (parse-fae `3))

(test (compile (parse `{+ 1 x}))
      (parse-fae `{+ 1 x}))

(test (compile (parse `{f {+ 1 2}}))
      (parse-fae `{f 3}))

(test (compile (parse `{- {+ 1 2} 3}))
      (parse-fae `0))
```

Constant Folding

```
(define (compile an-fwae)
  (type-case FWAE an-fwae
    ...
    [W-add (l r) (try-constant-fold
                  (add (compile l)
                       (compile r)))]
    [W-sub (l r) (try-constant-fold
                  (sub (compile l)
                       (compile r)))]
    ...))
```

Any time we see an **add** or **sub**

See if we can constant fold

Constant Folding

```
(define (try-constant-fold an-fae)
  (type-case FAE an-fae
    [add (l r)
         (if (and (num? l) (num? r))
             (num (+ (num-n l) (num-n r)))
             an-fae)]
    [sub (l r)
         (if (and (num? l) (num? r))
             (num (- (num-n l) (num-n r)))
             an-fae)]))
```

- Know which language you're operating on!
  - We go after the translation, so **`FAE`**
- Our implementation happens to be interleaved with translation
  - So get recursion and nesting for free
  - But could do as separate, standalone translation pass