Lesson 20
COMPILERS

August 1, 2023

Lesson Plan

1 - The History of Programming Languages

Prologue

Let me tell you a story.

Once upon a time, programming languages weren't real.

They were an idea in someone's head, of something that might be, but which had never been done yet.

There weren't IDEs, there weren't VS Code extensions, there weren't syntax highlighters, and there was no Standard ML.[1]

[1] A true horror story.

Early Computer Hardware

λ

Actually, once upon a time, computers weren't real either.[2]

People did everything by hand or by simple machine. There were no calculators, and there was no Twitter.[3]

An early invention in the 1800s led to a loom which could weave a certain pattern based upon usage of punched cards, literally cards with holes in them. This was one of the first examples of what could be considered programming, albeit strictly for loom-weaving.

A little later on in the 1800s, Ada Lovelace and Charles Babbage worked on a theoretical proposal of a machine called the **Analytical Engine**, that could execute arbitrary computations. This was the advent of the general purpose computer.

[2] I'm less bothered about that one.
[3] Soon, there might not be again.

Early Computer Hardware

By the beginning of the 1900s, Herman Hollerith had the idea of being able to feed data into a machine via use of punch card, for the purposes of data processing, record-keeping, and basic arithmetic operations.

He would found the Tabulating Machine Company on this premise, primarily using punched cards for the purpose of data storage. He then would realize this was a stupid name, and rename it to International Business Machines Corporation, now known as IBM.

Fast forward a couple of decades, and Konrad Zuse comes out with the Z3, a working, programmable electrical computer. Punched cards were used both to submit program code to the machine, as well as to store the data that it kept and output.

Punch Card Computing

Some people have expressed that programming in SML/NJ can be an unpleasant experience, due to a lack of ecosystem and documentation, and rather interesting error messages.

I want to put this scenario into your head.

You are an engineer in the mid 1900s, and you are one of the first programmers in the world. You painstakingly punch characters, hole by hole, into punch cards, and then gather your cards that comprise your program and then walk across the hall to the other room, where you can wait in line to submit a request to run your program.

If you make a mistake while punching a card, you have to start over. If you drop your deck of cards, the program is gone.

And still, programming languages are not real.

Assembly Programming

In the 1940s, programs were now being written in assembly language, the lowest form of language understandable by computers. These are essentially just blocks of bits that have some particular meaning to the computer, bijected with English words so that we can remember what they do.
For instance, the following might be an assembly-like syntax:

```
ADD  r1  r2          (* add r1 to r2 *)
CMP  res  0          (* compare res to 0 *)
JZ    0x00067AB3EF (* if zero, jump somewhere *)
GOTO 0x000B72AF48 (* otherwise jump somewhere else *)
```

I quote Wikipedia here on the topic, as follows:

> *It was eventually realized that programming in assembly language required a great deal of intellectual effort. [citation needed]*

Wikipedia claims that a citation is needed. I think that citation is only needed for people who have never had to write assembly language before.

Assembly Programming

So programmers spend hours, days, weeks poring over these very basic symbols, having to reconstruct every single operation that the computer is doing in their heads, and needing to remember the exact state of the computer at each step, at the risk of making an incorrect assumption and writing a bug.

There is no recovery from a typo. There is no such thing as a type error.

And still, programming languages are not real.

A Language for Computing

Then, one day, someone has an idea. Maybe multiple people have an idea.

One such person was John W. Backus, in the 1950s, whose idea was that maybe we don't need to write assembly by hand. Maybe we can write instructions that could then be converted into the assembly itself.

John Backus called the idea in his head FORTRAN, and by the end of the decade, a program was implemented to do exactly that.

This was the first FORTRAN compiler,

And then, programming languages were real.

2 - Compiler Theory

What is a Compiler?

What on earth really is a compiler?

Def A **compiler** can be put in very broad terms as a **program that translates data from one form to another**. Usually, we refer to a compiler for programming languages, which turns text in some language into some other form.

For instance, SML/NJ is a compiler which takes in SML programs, and turns it into native machine code for your computer, for a variety of computer architectures.

A sister idea is also that of:
Def An **interpreter** is a program which reads in a program, and executes it on the computer directly, without necessarily explicitly translating it to a different form.

The SML/NJ REPL is an interpreter.

Compilers and Interpreters

Put in SML terms, we can come up with a type signature for compilers and interpreters:

```
(* SML text -> assembly language text *)
val compile : string -> string
(* a function which executes some assembly code directly.
 * all computers come with this prepackaged. *)
val run : string -> unit

(* execute SML text directly *)
val interpret : string -> unit
```

In a broad sense, we should have that `interpret` $\cong$ `run o compile`. Put simply, interpreting a program is the same as compiling it first, and then running it.

Compilers and Interpreters

A way to understand interpreters and compilers comes from natural language. For instance, I am both a compiler and an interpreter.

I am a compiler, because I have a rudimentary understanding of Mandarin Chinese. This means that when I hear Mandarin, I can translate it into English first, and then understand the resulting English.

I am also an interpreter, because I natively speak English. When I hear English, I immediately understand its meaning, without needing to mentally translate it.

Circuitous Implementation

One idea that always is hard to accept is the fact that a programming language can be implemented in itself.

For instance, PyPy is a Python implementation written in Python. Similarly, the SML/NJ compiler is written in SML, and the Rust compiler is written in Rust.

Why can this happen?

Let's use the shorthand of $\texttt{compile}_{\langle\text{lang}\rangle}$ to denote a function of type `string -> string`, which takes in code written in the language ⟨lang⟩, and outputs assembly instructions.

Programming Languages as Ideas

Understand that a programming language originates as just an *idea*.

Before John Backus actually implemented FORTRAN, he had the idea in his head that there should be some high-level language that he could program in. He possibly had the syntax in his head, and called it FORTRAN, but a program written in FORTRAN couldn't do anything, because no computer could understand FORTRAN at the time.

But, luckily enough, we had assembly and punch cards. This meant that it was already possible to compute – in particular, it was possible to tell a computer how to do *anything*, how to compute any computable function.

One such function of interest is the $\texttt{compile}_{\text{FORTRAN}}$ function, or in other words, a FORTRAN compiler. This happens to be a concrete instance of a function which is computable, so it is *possible* to use assembly and punch cards to write it.

Bootstrapping Compilers

But, once we now have $\texttt{compile}_{\text{FORTRAN}}$, it is possible to write FORTRAN code and actually turn it into machine instructions, and run it.

Therefore, it is possible to write FORTRAN code which computes anything.

In particular, it is still possible to write a function $\texttt{compile}_{\text{FORTRAN}}$, except now instead of being written in assembly language, it is written in FORTRAN itself, and given meaning by the existing FORTRAN compiler.

This is an example of what is called **bootstrapping** in compiler theory. We can implement a language in itself, but only so long as there already exists a compiler which can understand that language.

Bootstrapping Compilers

So, PyPy and SML/NJ and the Rust compiler all work the same way.

Guido van Rossum started with an idea in his head of a language called Python, with a particular syntax and a particular kind of evaluation model. It didn't exist yet, but it was an idea in his head.

Then, he took his language of choice, and then implemented a function $\texttt{compile}_{\text{Python}}$ in it. This was the first Python compiler [4], and made it possible to write and run Python code.

Once that was done, now we could write any function, including *rewriting* $\texttt{compile}_{\text{Python}}$, but this time in Python. This was then called `pypy`.

[4] Actually, Python is an interpreted language, so at this point there wasn't yet a Python "compiler". But for the sake of the story, interpreters are similar enough.

Compilers from 5000 Feet

Now that we've set the stage for what compilers are, and for what they do, we can talk about the actual implementation of a compiler.

Recall that a compiler, such as $\texttt{compile}_{\text{SML}}$, is written in some language, and has type `string -> string`, where it takes in the text of an SML program and returns an executable.

A compiler achieves this goal by translating the input program through a series of intermediate forms, simplifying and optimizing along the way, until finally producing an assembly program.

A meta question comes to mind, which is – why are we studying compilers in a course on functional programming?

Even the most dogged, determined hater of functional programming cannot deny that functional programming is incredibly suited towards writing compilers.

The killer application for functional programming is in writing compilers, because compilers are just transformations on data. Functional programming mediates that relationship by enforcing typed guarantees on that data, as well as offering fundamental constructs (pattern matching, datatype declarations) that make the entire process an absolute joy.

For the rest of this presentation, we will assume we are trying to implement a toy SML compiler in SML.

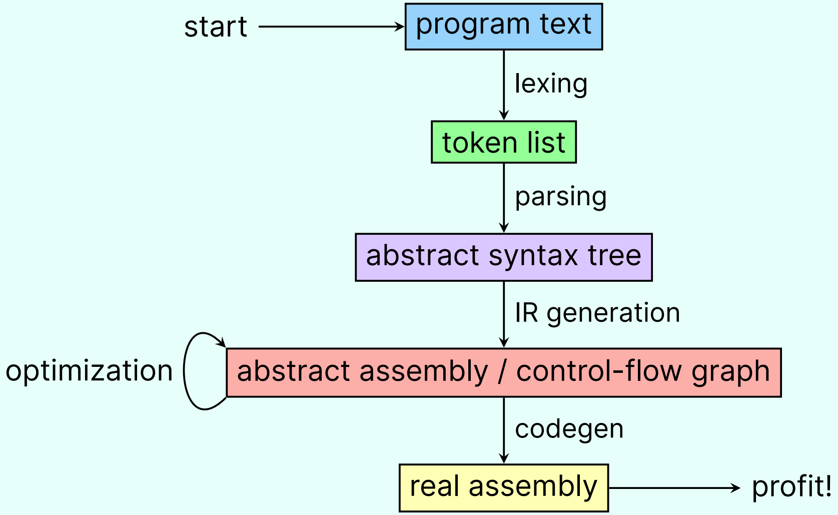
3 - Compiler Implementation

The Lifecycle of a Compiler

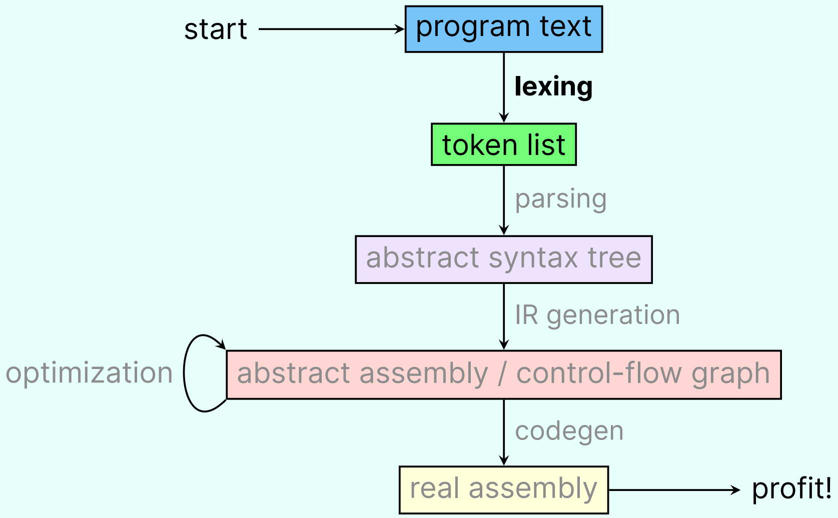
Most all compilers follow the same structure:

- **lexing**, which takes in a `string` program, and outputs a `token list`, which simply groups together fundamental units of the program
- **parsing**, which takes in a `token list`, and outputs an **abstract syntax tree** of type `ast`, which is a tree representing the program's structure
- **intermediate representation**, which turns the abstract syntax tree into **abstract assembly**, that breaks apart the high-level constructs into assembly-like primitives
- **optimization**, which tries to fine-tune the abstract assembly to be as performant as possible
- **code generation**, which involves turning the abstract assembly into real assembly

The Lifecycle of a Compiler: Lexing



Steps of a Compiler



An Example Compilation

Let's visually see how we can think about a compiler. We'll start with an example SML program, of type `string`.

```
val x = 2 - 1

fun foo (y : int) = 5 + x
```

Tokenization

We then **tokenize** the input program, so that instead of thinking of it as a list of characters, we group together all characters that are part of the same semantic unit.

Linguistically, this is similar to reading sentences as words, instead of as a list of letters. Let’s highlight all the "words" of this program.

```
val x = 2 - 1

fun foo (y : int) = 5 + x
```

SML is a whitespace-agnostic language, so we don’t actually care about the whitespace here. It just serves to separate distinct tokens.

Now, let’s turn it into a list of tokens, annotated with each token’s meaning.

Tokens as a Datatype

In SML, we could define a type `token` which simply denotes each "word" of a program. It might look something like this:

```
datatype token =
  (* data *)
    ID of string | NUM of int

  (* keywords *)
  | VAL | FUN | TYPE (* ... *)

  (* syntax *)
  | LPAREN | RPAREN | COLON | EQ | PLUS | MINUS
  (* ... *)
```

Tokens, Pictorially

So after tokenization into the `token` type we just defined, we might get something like this:

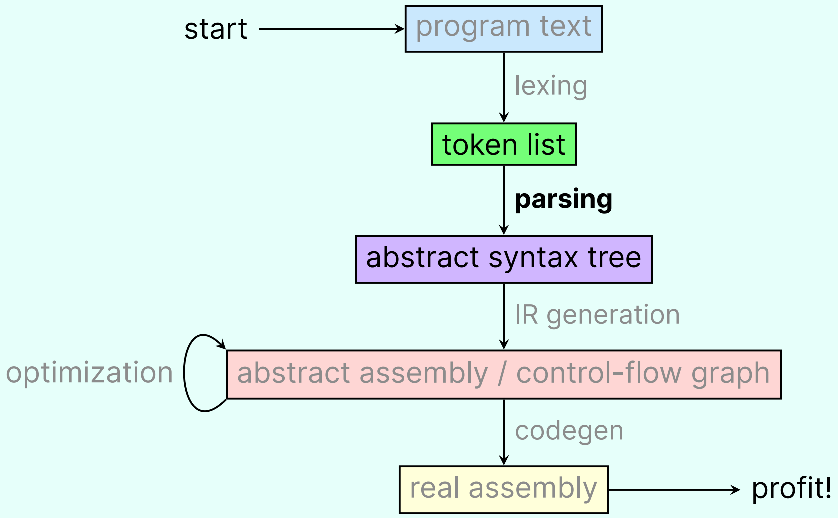
val	x	=	2	-	1
VAL	ID	EQ	NUM	MINUS	NUM

fun	foo	(	y	:	int	)	=	5	+	x
FUN	ID	LPAREN	ID	COLON	ID	RPAREN	EQ	NUM	PLUS	ID

Or, written as actual SML code:

```
[ VAL, ID "x", EQ, NUM 2, MINUS, NUM 1]
@ [ FUN, ID "foo", LPAREN, ID "y", COLON, ID "int", RPAREN ]
@ [ EQ, NUM 5, PLUS, ID "x"]
```

The Lifecycle of a Compiler: Parsing



Recursive Problems, Recursive Programs

One thing we note before proceeding is the fact that programs are naturally recursively defined. We can witness this by the fact that all the following are programs:

```
if true then big else ()
```

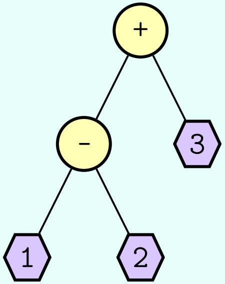
```
if true then if true then big else () else ()
```

```
if true then if true then if true then big else () else ()
   else ()
```

Towards Abstract Syntax

It turns out this makes programs a prime candidate for a recursive `datatype` declaration!

By analogy, if you are familiar with the idea of **op trees**, recall that we can have a tree corresponding to some arithmetic expression:



This tree happens to denote the expression `(1 - 2) + 3.`

Abstract Syntax is Abstract

This is just **abstract syntax**, though, since it elides some of the specific syntactic details, like the fact that there is a left and right paren around the subtraction.

In the end, this doesn't matter, because the tree structure serves as a proxy for what the parentheses were trying to tell us. We thus can get away from the precise coding details, while preserving the meaning, by using an **abstract syntax tree**, or AST for short.

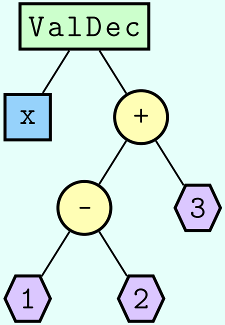
Programs are Trees

Well, we can do something very similar to op trees with programs. We will instead have an **abstract syntax tree** which denotes the structure of the program.

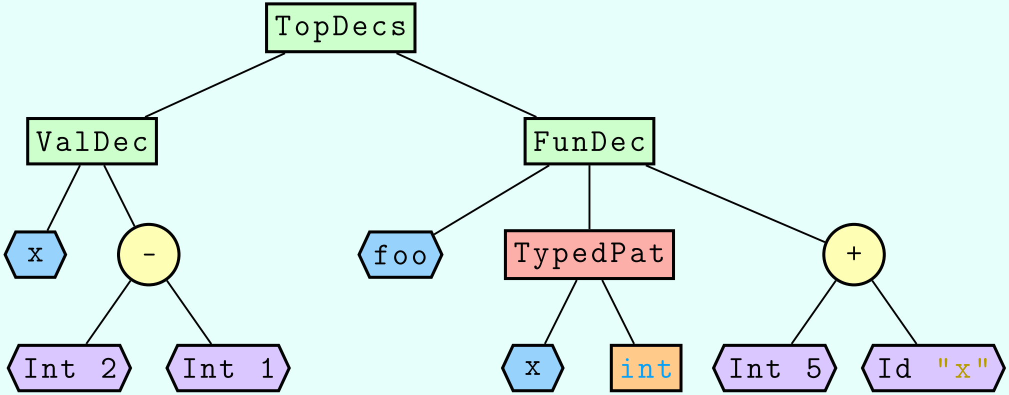
This tree denotes the program

```
val x = (1 - 2) + 3
```

Note how it has no mention of parens or the = sign, because they don't actually matter in terms of what the program *means*!
Generally, we can get rid of things like colons, equals signs, keywords, and parentheses in abstract syntax. These syntactic details only existed to let us know what the actual underlying tree looked like.



So, for our running example program, we could obtain the following abstract syntax tree:



Abstract Syntax as a Datatype

The SML code for how we might represent an abstract syntax tree looks similarly to the C_{not} problem from your homework. We could write:

```
datatype exp =
    Int of int
  | Id of string
  | Plus of exp * exp                       (* e1 + e2 *)
and declaration =
    ValDec of pat * exp                     (* val <pat> = <exp> *)
  | FunDec of string * pat list * exp
      (* fun <id> <p1> ... <pn> = <e> *)
  | (* ... *)
and pattern =
    IdPat of string
  | TuplePat of pattern list                (* (p1, p2, ..., pn) *)
  | (* ... *)
```

Recursive Descent Parsers

Usually, an abstract syntax tree can be obtained from a list of tokens via a straightforward **recursive-descent parser**.

Def A **recursive-descent parser** is one comprised of many mutually recursive functions, each of which simply is responsible for parsing a single construct (such as expressions, patterns, declarations) from a list of tokens.

So for instance, we might have a bunch of functions, such as:

```
val parsePat : token list -> pat * token list
val parseExp : token list -> exp * token list
val parseDec : token list -> dec * token list
```

which parse a tree corresponding to a pattern, expression, or declaration from the front of a `token list`, and then return the rest of the `token list`. This should feel like the regex matcher!

The SML code for that might look like:

```
fun parseDec (ts : token list) : dec * token list =
  case ts of
    (* val <pat> = <exp> *)
    VAL::ts2 =>
      let
        val (pat, ts2) = parsePat ts
        val ts3 = expect EQ ts2
        val (exp, ts4) = parseExp ts3
      in
        (ValDec (pat, exp), ts4)
      end
  | FUN::ts2 => (* ... *)
  | TYPE::ts2 => (* ... *)
  | (* ... *)
```

Regions of a Parse Tree

We can see how it works by going back to our `token list` from earlier.

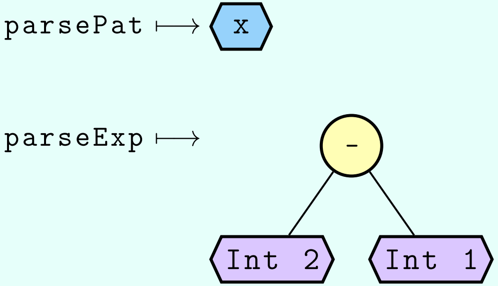
Using the `parseDec` function, we essentially split the token list into regions based on whether they correspond to patterns or expressions or related:

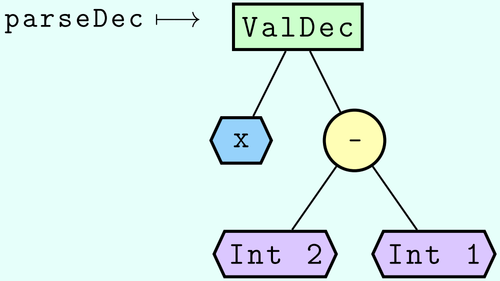
$$\overbrace{\substack{\texttt{val}\\\texttt{VAL}}\quad \underbrace{\substack{\texttt{x}\\\texttt{ID}}}_{\texttt{parsePat}}\quad \substack{\texttt{=}\\\texttt{EQ}}\quad \underbrace{\substack{\texttt{2}\\\texttt{NUM}}\quad \substack{\texttt{-}\\\texttt{EQ}}\quad \substack{\texttt{1}\\\texttt{NUM}}}_{\texttt{parseExp}}}^{\texttt{parseDec}}\quad \overbrace{\substack{\texttt{fun}\\\texttt{;}}\quad \substack{\texttt{foo}\\\texttt{ID}}}^{\text{(rest of token list)}}$$

Regions of a Parse Tree

The highlighted regions then return to us subtrees of certain types, as the return values of `parsePat` and `parseExp`.

Then, `parseDec` just needs to string them together, and place them under the constructor `ValDec`, corresponding to a node of the syntax tree representing a `val` declaration.





On Implementing Parsing and Lexing

I have so far been brief on how you implement lexers and parsers.

If you look at the code above for the `parseDec` function, it is reasonably boring-looking. The code mostly follows the structure of the program.

Lexing and parsing are generally considered to be "solved" problems, in the field of computer science, due to decades of advances in lexing and parsing theory. It turns out that lexers and parsers are so straightforward that many lexers and parser nowadays are no longer written by humans, but actually autogenerated from a specification.

So they are generally not super important. I enjoy writing parsers, though. [5]

[5] Such generators are called **lexer and parser generators**. You could also call them **lexer compilers** and **parser compilers**, confusingly.

Playing with Program Trees

λ

Once you have ASTs, everything interesting can happen.

ASTs essentially encode all the essential syntactic structure of a program, while getting rid of all of the fat that is not strictly necessary to understand what a program *means*.

It is at this point we can write transformations and analyses on those ASTs, to imbue the tree with actual meaning.

Key Seen in this way, compilation is nothing more than transformation and operations on trees.

Playing with Program Trees

For instance, something very pivotal that occurs soon after you obtain a syntax tree is that you can type-check a program. The code for that might look like:

```
datatype ty = IntTy | StringTy | RealTy | (* ... *)

fun typecheck (e : exp) : ty =
  case e of
    Int _ => IntTy
  | Plus (e1, e2) =>
      (case (typecheck e1, typecheck e2) of
        (IntTy, IntTy) => IntTy
      | (RealTy, RealTy) => RealTy
      | _ => raise Fail "expected pair of ints or reals"
      )
  | (* ... *)
```

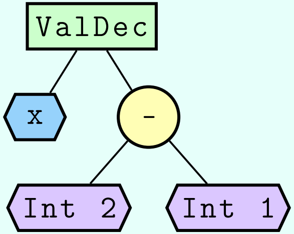
We see it's just a straightforward recursive function on a tree.

Compiler Optimizations

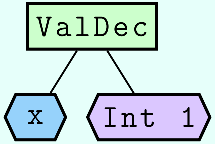
Type-checking is just one operation you can perform on an AST.

Another critical part of compilers is **compiler optimizations**, which is the problem of turning an AST into a *semantically equivalent* AST that is simpler in some form.

For instance, consider the sub-tree of our running program, corresponding to the declaration of `x`:



However, on inspection, it might as well be:



Constant Folding

This is a classic compiler optimization called **constant folding**.

Def **Constant folding** is a compiler optimization where the AST is simplified to evaluate any constant arithmetic expressions at compile time, simplifying the AST.

It may seem fairly trivial to simplify expressions like `2 - 1`. After all, this takes a fairly miniscule amount of time at runtime! Just remember:

- **Optimizations add up**. Being able to simplify an expression might lead to further simplifications, especially nearer to assembly, when pretty much everything just becomes arithmetic operations.
- **Computers repeat themselves**. Computers often run the same routine many, many times, so small things like less addition operations can lead to a huge difference, when it's happening billions of times.
- **Less operations, less bloat**. Less operations lead to smaller file sizes and faster execution. If I compile the expression `2 - 1`, I need to issue multiple instructions, versus if I just have `1`.

Constant Folding

It can be implemented simply via the following recursive function:

```
fun cfold (e : exp) : exp =
  case e of
    Plus  (Int i1, Int i2) => Int (i1 + i2)
  | Minus (Int i1, Int i2) => Int (i1 - i2)
  | Div   (Int i1, Int i2) => Int (i1 div i2)
  | _ => e
```

At least, it would be, if this wasn't incredibly wrong.

Constant Folding?

This code seems reasonable. If we are trying to fold the constant integer or an identifier, we can't do anything, so we return just that expression.

Otherwise, if we are folding an arithmetic operation of some literal integer arguments, we simplify the tree to that operation's result. However:

Error 1: **This program can crash.**

```
fun cfold (e : exp) : exp =
  case e of
    Plus  (Int i1, Int i2) => Int (i1 + i2)
  | Minus (Int i1, Int i2) => Int (i1 - i2)
  | Div   (Int i1, Int i2) => Int (i1 div i2)
  | _ => e
```

Constant Folding, v1

Our program will crash upon trying to compile `val x = 1 div 0`!

Compiling is just a staged version of interpreting code – it is meant to produce something which can *then* be run, but a compiler definitely cannot crash or loop depending on its input program's behavior. So this is a big no-no in compilers.

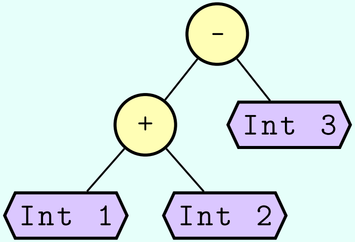
We need to fix it like so:

```
fun cfold (e : exp) : exp =
  case e of
    Plus  (Int i1, Int i2) => Int (i1 + i2)
  | Minus (Int i1, Int i2) => Int (i1 - i2)
  | Div   (Int i1, Int 0) => e
  | Div   (Int i1, Int i2) => Int (i1 div i2)
  | _ => e
```

Constant Folding, v2

Error 2: our constant folding function doesn't recurse!

This means if the expression to be folded is not literally at the top of the expression, it won't happen, for instance for the following tree:



So, let's fix our code again:

Constant Folding

```
fun cfold (e : exp) : exp =
  case e of
    Plus  (Int i1, Int i2) => Int (i1 + i2)
  | Minus (Int i1, Int i2) => Int (i1 - i2)
  | Div   (Int i1, Int 0)  => e
  | Div   (Int i1, Int i2) => Int (i1 div i2)
  | Plus  (e1, e2) => Plus  (cfold e1, cfold e2)
  | Minus (e1, e2) => Minus (cfold e1, cfold e2)
  | Div   (e1, e2) => Div   (cfold e1, cfold e2)
  (* ... many more cases ... *)
```

Unfortunately, this means we will need a case for every single constructor in the `exp` type.[6]

[6] There are techniques in functional programming that mean this can actually be written much more tersely than this, without all the boilerplate.

Compile-Time Optimizations

Constant folding is nice, but in practice usually there's not a whole lot of compile-time evaluatable expressions in the actual source of the code.

This would require someone to write something like

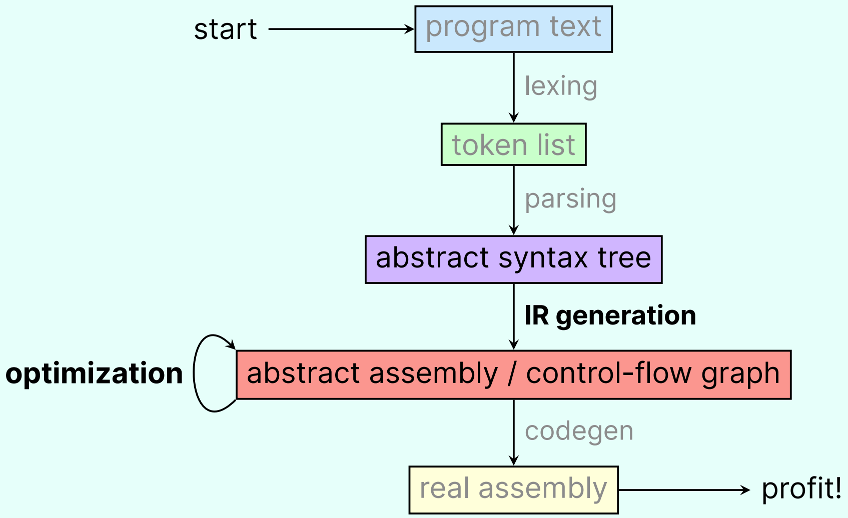
```
val x = 1 + 2
```

in their source code.[7]

For things like pattern matching, function calls, loops, and complex data structures, it's difficult to optimize, because these structures are too high-level. For most optimizations, we need to descend closer to assembly language.

[7]Which happens, and sometimes by some of you, but not often.

The Lifecycle of a Compiler: IR Generation



Assembly, but Abstract

For this purpose, we have **abstract assembly**, which is a kind of primitive code that looks like assembly language, but doesn't need to touch things like registers.

Such instructions break apart the nested structure of programs to achieve a very simple layout. For instance, here's how we might translate the following function: [8]

```
def f (x, y, z):
  return x + (y + z)
```

```
t1 <- y + z     (* a temp variable *)
t2 <- x + t1    (* another temp *)
ret t2          (* return the result *)
```

Here, we do operations on step at a time – no nesting.

[8] It's actually quite difficult to compile SML, for a number of reasons that have to do with the nice things we have enjoyed so far in this course. For the rest of the lecture, we will assume a pseudo-Python language, which is easier to explain.

Code Without Brakes

This is fine for what is known as **straight-line code**, which is code that doesn't perform any branches of control flow, or jumps in program logic.

Unfortunately, both of these things are quite common in programming languages. To that end, we translate abstract syntax into a **control-flow graph**, or CFGs, which links blocks of straight-line code (called **basic blocks**) by pointers which denote which blocks can reach each other.

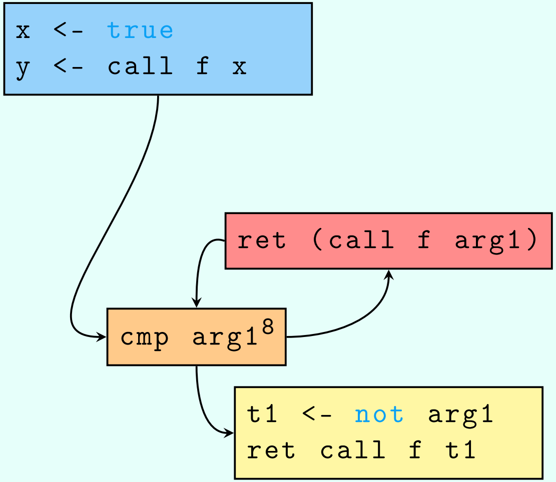
```
def f(x):
  if x:
    return f(x)
  else:
    return f(not x)

x = true
y = f(x)
```

A Controlled Flow Example

```
def f(x):
  if x:
    f(x)
  else:
    f(not x)

x = true
y = f(x)
```

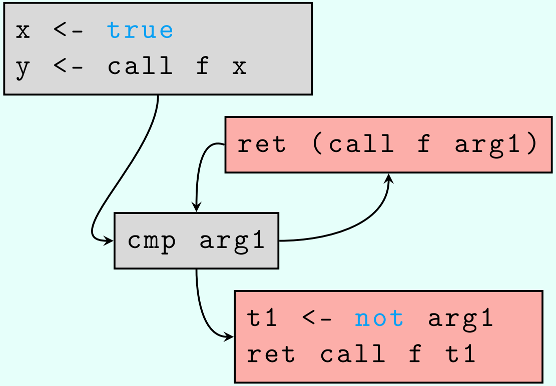


[8]When I write `cmp`, in reality it's quite a bit more complicated. There's logic to handle certain kinds of comparisons, which thing you should jump to... just let the picture guide you. We can elide those and just assume we pick the right edge.

A Cyclic Graph

But, what's something that we notice about this CFG?

No matter which way we go, we always end up at a call to `f`!



```
x <- true
y <- call f x
```

```
ret (call f arg1)
```

```
cmp arg1
```

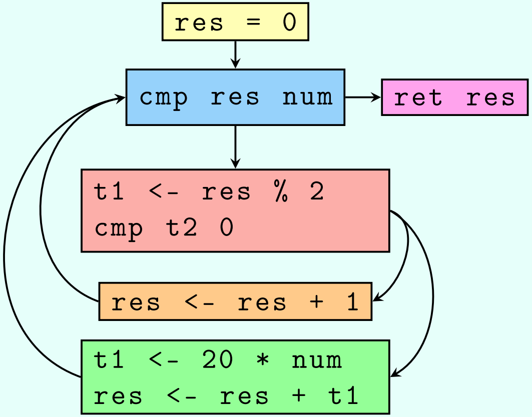
```
t1 <- not arg1
ret call f t1
```

In other words, a surefire infinite loop.

An Unoptimal Example

Control-flow graphs also make it apparent when optimizations *cannot* be done! For instance, consider the following program:

```
def f(x, num):
  res = 0
  while (res < num):
    if res \% 2 == 0:
      res += 1
    else:
      res += (20 * num)
  return res
```



Unoptimization

What can we say about our ability to optimize out the computation of `20 * num`?

```
def f(x, num):
  res = 0
  while (res < num):
    if res \% 2 == 0:
      res += 1
    else:
      res += (20 * num)
  return res
```

Unoptimization, v1

We could try to insert it out of the function, but now we don't have access to the `num` variable.

```
temp = 20 * num

def f(x, num):
  res = 0
  while (res < num):
    if res \% 2 == 0:
      res += 1
    else:
      res += temp
  return res
```

Unoptimization, v2

We could try to put it outside of the loop, but it's possible we always enter the highlighted `true` case, and thus now we have made the program slower.

Statically, we have no idea whether we enter the true or false case, because we don't have the input values to the function.

```
def f(x, num):
  res = 0
  temp = 20 * num
  while (res < num):
    if res \% 2 == 0:
      res += 1
    else:
      res += temp
  return res
```

Unoptimization, v3

Even if we had a good guarantee that we enter the `true` case enough for it to be worth the optimization, what if instead of computing `20 * num`, it was `20 div num`?

Then, we introduce additional unsafe behavior!

```
def f(x, num):
  res = 0
  temp = 20 div num
  while (res < num):
    if res \% 2 == 0:
      res += 1
    else:
      res += temp
  return res
```

Safe Optimizations

Compiler optimization is an interesting field because we have to try very hard to preserve an equivalent program, and we can only do that if we do quite a bit of thinking to find the cases where this optimization is safe – in essence, that it will not change the behavior of the program.

Check your understanding Is it true that you can optimize forward `20 * num` if that expression appears on all boxes onward?

In practice, this desugars to the content of our next lecture, program analysis, and involves a technique known as **dataflow analysis**, which guarantee that we can obtain that information in finite time.

Assembly Optimizations

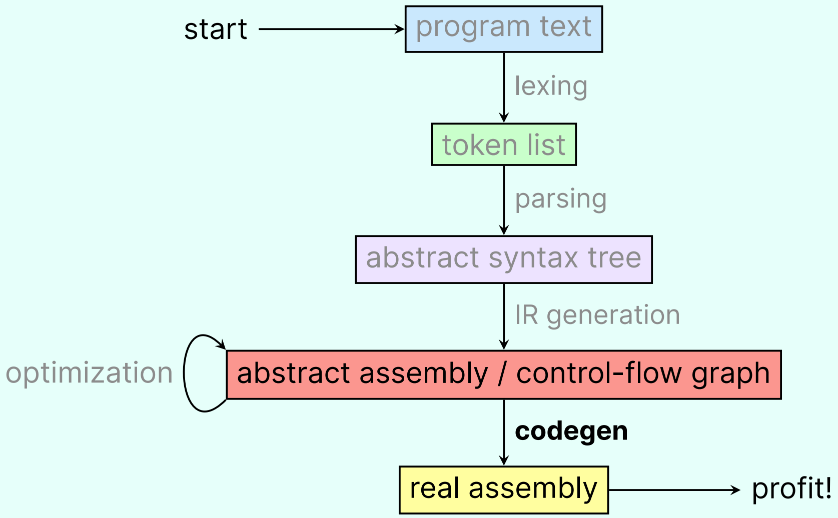
With CFGs, you can do easy optimizations, because the flow of the program is very apparent. This leads us to a whole host of possible optimizations, the list of which is far too comprehensive to go over here.

The basic idea is that these are two main kinds of optimizations:

Def **Local optimizations** are optimizations which run purely within a single basic block of straight-line code.
Some examples of these include constant folding and copy propagation.

Def **Global optimizations** are optimizations which run interprocedurally, that is, by considering the flow of data through multiple basic blocks or functions.
Some examples of these include dead code elimination, unused variables, and function inlining.

The Lifecycle of a Compiler: Code Generation



Generating Assembly

Once the control flow graph has been simplified to satisfaction, we can then move on to the step of generating actual assembly instructions from the abstract assembly.

The primary difference is that the abstract assembly assumes that you can move values to and from an infinite array of temporary variables.

Unfortunately, actual computers exist in the real world, and do not have an infinite amount of places you can put data. This isn't a big deal always, though, because we don't need to keep all data around forever.

This means that essentially, a compiler must have future sight, and perfectly plan out everywhere that it puts its data during the program's run. This problem is called **register allocation**, and is far beyond the scope of this course, and my ability to explain right now.

An Analogy for Register Allocation

Consider the following analogy.

I am really popular. I have lots of friends.[9]

Unfortunately most of them hate each other and I only have 8 seats at my birthday party, which is happening at Chuck E. Cheese's.[10]

I want as many of them to come, but they are `all` only `available` at certain times, and they'll be cranky if they can't be there for the full time they're available.

How can I plan a precise schedule with the time that each friend is allowed to come, to maximize their (and my) happiness?

[9] Haters will say I wrote this analogy purely so I could put this into my slides.

[10] Where a kid can be a kid.

Arriving at Executables

Then, finally, real assembly is generated from the abstract assembly, and the executable is fit to run. The story is over.

This is the bird's eye view of a compiler. There are many facets to each phase that were not elaborated upon, but in terms of the intuition behind what a compiler is and does, this should suffice.

The key thing to realize here is that at the end of the day, compilers are not magic!

Functional Programming and Compilers

Functional programming ultimately fits compilers well for a few reasons:

- **Compilers are fancy tree transformations.** Functional languages are very good at dealing with recursively defined structures which look like trees.
- **Compilers cannot be wrong**. If you compile your code, and the resulting code does not do what it should, this is the doomsday scenario. If we cannot trust the compilers we use, then there is no hope for programming. Functional programming is safer in general, leading to less silly mistakes and unsafe behavior.
- **Compilers are pure**. When translating a compiler from one stage to another, we generally expect deterministic results. Thus, no mutability needed.

Thank you!