(BNF). Each line of this grammar is a *production rule* that defines how a language construct can be formed from a sequence of other language constructs and tokens. Every symbol that appears on the left-hand side of a production rule (like `<function>`) is called a *non-terminal symbol*. Individual tokens, like keywords, identifiers, and punctuation, are called *terminal symbols*. All non-terminal symbols are wrapped in angle brackets, and specific tokens (like `;`) are wrapped in quotes. The `<identifier>` and `<int>` symbols are special. They represent individual identifier and constant tokens, respectively, but these tokens aren't set strings like the other terminal symbols. Since there's not an easy way to define those symbols in Backus-Naur form, we describe each of them using a *special sequence*—that's just a plain English description of the symbol, wrapped in question marks.

Listing 2-6 looks a lot like the AST definition in Listing 2-5. In fact, it has exactly the same structure—every AST node in Listing 2-5 corresponds to a non-terminal symbol in Listing 2-6. The only difference is that Listing 2-6 specifies exactly which tokens we'll find at each node of the tree. This helps us figure out when we need to start processing a new node at the next level down in the AST, and when we've finished processing a node and can go back up to its parent on the level above.

Just like later chapters will introduce multiple constructors for some AST nodes, they'll introduce multiple production rules for the corresponding symbols. For example, here's how we'll add a production rule for `<statement>` to support if statements:

```
<statement> ::= "return" <exp> ";" | "if" "(" <exp> ")" <statement> [
"else" <statement> ]
```

Note that brackets in BNF indicate that something is optional, just like questions marks in ASDL.

You'll need to refer to this formal grammar while writing the parser, but you don't need to explicitly define the grammar rules anywhere in your compiler.

Recursive Descent Parsing

Now that you have an AST definition and a formal grammar, let's talk about how to actually write the parser. We'll use a straightforward technique called *recursive descent parsing*. A recursive descent parser uses a different function to parse each non-terminal symbol and return the corresponding AST node. For example, when the parser expects to encounter the `<statement>` symbol we defined in Listing 2-6, it will call a function to parse that symbol and return the `statement` AST node we defined in Listing 2-5. To parse an entire program, you'll call the function that parses the `<program>` symbol. With each function call to handle a new symbol, the parser descends to a lower level in the tree. That's where the *descent* in recursive descent comes from. (It's called *recursive* descent because the grammar rules are often recursive, in which case the functions to process them will be too. For example, the operand of an expression could be another expression—we'll see an example of that in the next chapter.)

Because the process of parsing a symbol is easier to explain in code, let's look at some pseudocode for parsing a `<statement>` symbol. Once you understand it, you can write your own code to process `<statement>` and all the other non-terminal symbols in Listing 2-6.

```
1parse_statement(tokens):
2    expect("return", tokens)
    return_val = parse_exp(tokens)3
4    expect(";", tokens)
5    return Return(return_val)

expect(expected, tokens):
    actual = take_token(tokens)
    if actual != expected:
        fail()
```

Listing 2-7 Parsing a statement

We'll call the `parse_statement` function 1 when we expect the list of remaining tokens to start with a `<statement>`. According to Listing 2-6, a `<statement>` consists of three symbols: the `return` keyword, an `<exp>` symbol, and a "`;`" token. First, we call a helper function, `expect` 2 , to verify that the first token really is a `return` keyword. If it is, `expect` just discards it so we can move on to the next token. If it isn't, there's a syntax error in the program. Next, the grammar tells us that the `return` keyword should be followed by an `<exp>` symbol. We need to turn this symbol into an `exp` AST node so we can construct the return statement. Since this is a different non-terminal symbol, it should be handled by a separate function, `parse_exp`, which I haven't defined here. Once we've gotten the AST node representing the return value back from `parse_exp` 3, we just need to verify that it's followed by the last token, a semicolon. We handle this with `expect` 4, just like we handled the `return` keyword at 2. At this point, we know the statement is syntactically valid, so we can return an AST node 5.

Note that the `parse_statement` function removes all the tokens that made up the statement from the `tokens` list. After `parse_statement` returns, its caller will keep processing the remaining tokens in `tokens`. If there are any tokens left after parsing the entire program, that's a syntax error.

The other thing to note is that this pseudocode is written a very imperative way. Functional languages (the sorts of languages I recommended in the last chapter!) generally won't let you modify the input list like I'm doing here. So the details of how your parser passes tokens around will probably differ from Listing 2-7, but the overall structure will be the same.

Right now, each symbol in our formal grammar has only one production rule. In later chapters, when some symbols have multiple production rules, your parser will need to figure out which production rule to use. It can do that by looking at the first few tokens in the list without removing them. Recursive descent parsers that look ahead a few tokens to figure out which production rule to use are called *predictive parsers.* The alternative to predictive parsing is *recursive descent with backtracking*—trying each production rule in turn until you find one that works.

Testing the Parser

Your parser should fail on the programs in *tests/chapter_2/invalid_parse* and succeed on the programs in *tests/chapter_2/valid.* To test the parser, run:

```
$ ./test_compiler /path/to/your_compiler --chapter 2 --stage
parse
```

This command only tests whether the parser succeeds or fails, so you may want to write your own tests to confirm that it produces the correct AST for valid programs and emits an appropriate error for invalid ones.

Implementation Tips

Write a pretty-printer. A pretty-printer is a function that prints out your AST in a human-readable way. This will make debugging your parser a lot easier. For example, a pretty-printed AST for the program in Listing 2-1 might look something like this:

```
Program(
  Function(
    name="main"
    body=Return(
      Const(2)
    )
  )
)
```

Give informative error messages. This will also help you debug your parser (and if anyone ever wants to use your compiler, it will help them too). An error message like `Expected ";" but found "return"` is a lot more helpful than `Fail`.

Writing the Code Generator

The code generation pass should convert the AST into x64 assembly. It should traverse the AST in roughly the order the program executes, producing the appropriate assembly instructions for each node. First, you need to define an appropriate data structure to represent the assembly program, just like you needed to define a data structure to represent the AST when you wrote the parser. You're adding yet another data structure, instead of writing assembly to a file right away, so that you can modify the assembly code after you've generated it. You won't need to rewrite any assembly in this chapter, but in later chapters you will.

I'll use ASDL again to describe the structure we'll use to represent assembly. Here's the definition:

```
1program = Program(function_definition)
2function_definition = Function(identifier name, instruction*
  instructions)
3instruction = Mov(operand src, operand dst) | Ret
4operand = Imm(int) | Register
```

Listing 2-8 ASDL definition of an assembly program

This looks a lot like the AST definition from the last section! In fact, this *is* an AST—but it's an AST that represents an assembly program, not a C program. Every node corresponds to a construct in assembly, like a single instruction, rather than a construct in C, like a statement. I'll refer to the data structure defined in Listing 2-8 as the "assembly AST" to distinguish it from the AST defined in Listing 2-5.

Let's walk through Listing 2-8. An assembly program still consists of a single function 1, which has a name and a list of instructions 2. The `*` in `instruction*` indicates that it's a list of instructions, not just one. The two instructions 3 that can appear in our very simple assembly programs are `mov` and `ret`, which we saw in Listing 2-2. The `mov` instruction has two operands: its copies the first operand, the source, to the second operand, the destination. The `ret` instruction doesn't have any operands. The two possible operands 4 to an instruction are a register and an *immediate value*, which is a value included in the instruction – in other words, a constant. For now, you don't need to specify which register to operate on, because your generated code will only use EAX. You'll need to refer to other registers in later chapters.

This pass will have a similar structure to the parser, so I won't write out the pseudocode here. You'll need a function to handle each type of AST node, which will call other functions to handle that node's children. Here's the equivalent assembly you need for each AST node:

Table 2-2 AST nodes and equivalent assembly

AST Node	Assembly Construct
`Program(function_definition)`	`Program(function_definition)`
`Function(name, body)`	`Function(name, instructions)`

`Return(exp)`	`Mov(exp, Register)`
	`Ret`

`Constant(int)`	`Imm(int)`

This translation is pretty straightforward, but there are a couple things to note. The first is that a single statement results in multiple assembly instructions. The second is that this translation only works if an expression can be represented as a single assembly operand. That's true right now, because the only expression is a constant integer. But it won't be true once we encounter unary operators in the next chapter. At that point, your compiler will need to generate multiple instructions to calculate an expression, and then figure out where that expression is stored in order to copy it into EAX.

Testing the Code Generator

To test the code generation stage, run:

```
$ ./test_compiler /path/to/your_compiler --chapter 2 --stage
codegen
```

The `--stage codegen` option will run your whole compiler. Like the equivalent `lex` and `parse` options, it will just check whether the compiler succeeds or fails. Your code generation stage should be able to handle any program that parses successfully, so this just tests that your code generation stage can handle every valid program. You may want to write your own tests to confirm that your compiler generates the assembly you expect.

Implementation Tips

Plan ahead for Part II! If you go on to do Part II of the book, you'll need to update the `Mov` instruction, and many of the other instructions we'll add in the next few sections, to store type information. You might want to define your assembly AST in a way that makes it easy to add more fields to each constructor later on. If there's no good way to do that in your implementation language, that's okay; it just means you'll have a little extra refactoring to do in part II.

Writing the Code Emitter

Now that your compiler can generate assembly instructions, the last step is writing those instructions to a file. Here's how to print each assembly construct:

Table 2-3 Formatting assembly

Assembly Construct	**Output**	
Top-level Constructs		
Program(function_definition)	(just print out the function definition)	

Function(name, instructions)		`    .globl <name>`
		`<name>:`
		`    <instructions>`

Instructions		
Mov(src, dst)		`    movl <src>, <dst>`
Ret		`    ret`
Operands		
Register		`%eax`
Imm(int)		`$<int>`

Note that there must be a line break between instructions, just like in Listing 2-2. The code emission step will need to traverse the assembly AST, just like the code generation stage traverses the AST from Listing 2-5. Because the assembly AST corresponds so closely to the final assembly program, the code emission stage will be very simple, even as you add more functionality to the compiler in later chapters.

NOTE If you're compiling on macOS, you need an underscore in front of the function name. For example, if you compile a function called `main`, the label in the resulting assembly should be `_main`. If you're on any other system, don't include an underscore.

Testing the Whole Compiler

To test the whole compiler from lexing to code generation, run:

```
$ ./test_compiler /path/to/your_compiler --chapter 2
```

This will compile each program in *tests/chapter_2/valid* with your compiler and GCC, run both executables, and verify that they produce the same exit code. It will also validate that all the invalid programs fail, but you should have already confirmed that during the earlier stages.

Implementation Tips

Generate readable assembly. When you debug your compiler, you'll spend a lot of time reading the assembly it produces. Your life will be easier if that assembly is nicely formatted. You can indent every line except for labels, like GCC does, to make your assembly more readable. (That's also how I've formatted Listing 2-2.) If you like, you can also include comments in your assembly programs. A `#` symbol in an assembly program comments out the rest of the line—it works just like `//` in C.

Summary

In this chapter, you wrote a compiler that can transform a complete C program into an executable that runs on your computer. You learned how to interpret a program written in x64 assembly, a formal grammar in Backus-Naur form, and an AST definition in ASDL. The skills and concepts you learned in this chapter—and the four compiler stages you implemented—are the foundation for everything you'll do in the rest of the book.

In the next chapter, you'll add support for unary operators to your compiler. Along the way, you'll learn about how assembly programs manage the stack, and we'll introduce a new way to represent the programs you compile to make them easier to analyze, transform, and optimize.

3

UNARY OPERATORS

C has several *unary operators*, which operate on a single value. In this chapter, you'll extend your compiler to handle two unary operators: negation and bitwise complement. You'll update the lexer, parser, and code emission pass to handle these new operators, but the code generation stage will require the biggest changes. Between parsing and code emission, you'll need to transform complex, nested expressions into simple operations that can be expressed in assembly. Instead of performing this transformation in a single compiler pass, we'll introduce a new intermediate representation between the AST produced by the parser and the assembly program produced by the code generation pass. We’ll also break up code generation into several smaller passes. To get started, let's look at a C program that uses our new unary operators, and the corresponding assembly we want to generate.

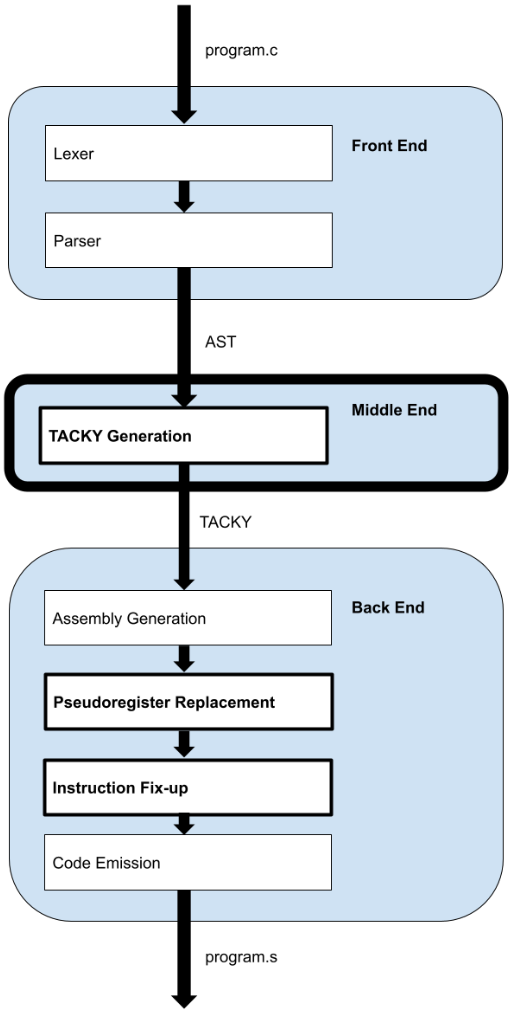

Figure 3-1 Stages of the compiler

Negation and Bitwise Complement in Assembly

In this chapter, you'll learn how to compile programs like this one:

```
int main() {
    return ~(-2);
}
```

Listing 3-1 A C program with negation and bitwise complement

This program uses both of the unary operations we'll introduce in this chapter. It also includes a nested expression. If you implement your compiler the way I suggest, it will produce the assembly listing below from the program in Listing 3-1:

```
    .globl main
main:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $8, %rsp
1   movl     $2, 2-4(%rbp)
3   negl     -4(%rbp)
4   movl     -4(%rbp), %r10d
5   movl     %r10d, -8(%rbp)
6   notl     -8(%rbp)
    movl     -8(%rbp), %eax
7   movq     %rbp, %rsp
    popq     %rbp
    ret
```

Listing 3-2 Assembly code for Listing 3-1

The first three instructions after `main` are the *function prologue*, which set up the current stack frame; I'll cover them more when I talk about the stack in detail below. After the function prologue, we'll calculate the intermediate result, -2, and then final result, 1, storing each of them at unique memory address. The resulting assembly isn't very efficient; we waste a lot of instructions copying values from one address to another. But this approach sets us up to generate more efficient assembly later on. In Part III, you'll see how to store as many intermediate results as possible in registers, instead of memory, which will speed things up and eliminate a lot of unneeded copies.

NOTE If you compile Listing 3-1 to assembly using GCC, or any other production C compiler, it won't look anything like Listing 3-2, because those compilers evaluate constant expressions at compile time, even when optimizations are disabled! The basis for this seems to be section 6.6 of the C standard, which states that "[a] constant expression can be evaluated during translation rather than runtime, and accordingly may be used in any place that a constant may be." Evaluating all constant expressions at compile time is an easy way to implement this part of the standard.

The first `movl` instruction 1 stores `2` at an address in memory. The operand `-4(%rbp)` 2 means "the value stored in the RBP register, minus 4." The value in RBP is a memory address on the stack (more on that below), so 2 refers to another memory address four bytes lower. That address is where instruction 1 will store `2`. Then we negate the value at this address with the `neg` instruction 3, so `-4(%rbp)` now contains the value `-2`. (Just like `mov`, `neg` has an `l` suffix to indicate that it's operating on a 32-bit value.)

Next, we need to handle the outer bitwise complement expression. The first step is copying the source value, stored in `-4(%rbp)`, to the destination address at `-8(%rbp)`. We can't do this in a single instruction, because the `mov` instruction can't have memory addresses as both source and destination operands. At least one operand to `mov` needs to be a register or an immediate value. We'll get around this by copying `-2` from memory into a scratch register, R10D 4, and from there to the destination memory address 5. We then take the bitwise complement of `-2` with the `not` instruction6, so memory address `-8(%rbp)` now contains value we want to return: `~(-2)`, which comes out to `1`. To return this value, we have to move it into EAX. The next three instructions make up the *function epilogue*, which tears down the stack frame and then returns from the function 7.

REPRESENTING SIGNED INTEGERS IN TWO'S COMPLEMENT

All modern computers use a *two's complement* representation of signed integers. A firm grasp on two's complement will help you understand and debug the assembly code your compiler generates. If you aren't already familiar with two's complement, or you need a refresher, here are a couple helpful resources:

"Two's Complement", by Thomas Finley, covers how and why two's complement representations work. (*https://www.cs.cornell.edu/~tomf/notes/cps104/twoscomp.html*)

The second chapter of *The Elements of Computing Systems*, by Noam Nisan and Shimon Schocken, covers similar material from a more hardware-focused perspective. This is the companion book for the Nand to Tetris project. This chapter is freely available at *https://www.nand2tetris.org/course*; click on the book icon under "Project 2: Boolean Arithmetic".

The Stack

There are still two unanswered questions about the code in Listing 3-2:

what the function prologue and epilogue do, and why we refer to stack addresses relative to a value in the RBP register. To answer both those questions, we need to talk more about the segment of program memory called the *stack*. The address of the top of the stack is stored in the RSP register, which is also called the *stack pointer*. (By convention, RSP points to the last used stack slot, rather than the first free one.) Like you'd expect with any stack data structure, you can push things onto the stack and pop values off of it; the `push` and `pop` assembly instructions do exactly that.

The stack grows towards lower memory addresses. When you push something onto the stack, you decrement RSP. Whenever I say "top of the stack", I mean the address stored in RSP, which is the lowest address on the stack. Note that the stack diagrams in this book, unlike other diagrams you may have seen, are oriented with lower memory addresses at the top. To help you remember how these diagrams are laid out, you can think of memory addresses like line numbers in a code listing: the top of the listing is at line 0, and line numbers increase as you go down. That means the top of the stack is on top of the diagram.

An instruction like `push $3` does two things:

1. Write the value being pushed (in this example, `3`) to the next empty spot on the stack. The `push` and `pop` instructions adjust the stack pointer in 8-byte increments, and the top value on the stack is currently at the address stored in RSP, so the next empty spot is RSP - 8.
2. Decrement RSP by eight. The new address in RSP is now the top of the stack, and the value at that address is `3`.

Figure 3-2 illustrates the effect of a `push` instruction on the stack and RSP register.

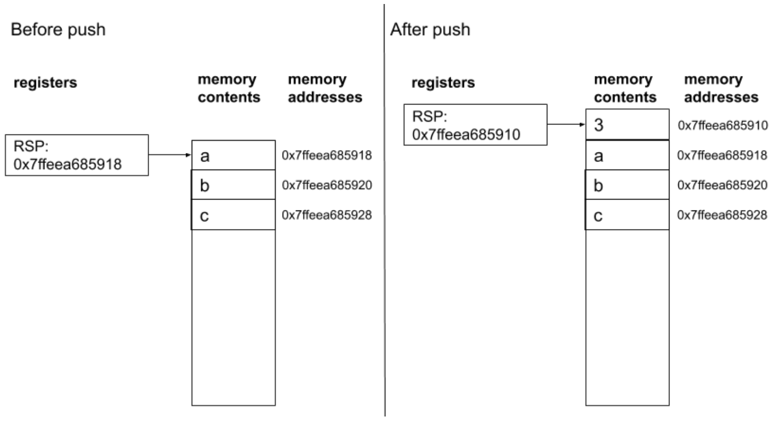

Figure 3-2 Effect of `push $3` instruction on memory and RSP.

The `pop` instruction does the opposite. For example, `pop %rax` would copy the value at the top of the stack into the RAX register, and then add eight to RSP.

Since the `push` instruction decrements the stack pointer by eight bytes, it has to push an 8-byte value. Likewise, the `pop` instruction can only pop 8-byte values off the stack. Because x64 memory addresses are eight bytes, we can use `push` and `pop` to put them on and take them off the stack. But the `int` type is only four bytes. If you want to put a 4-byte value on the stack, like the literal `2` from Listing 3-1, you can't use `push`, so you have to use `mov` instead. (On 32-bit architectures, the reverse is true; you can push and pop 4-byte values but not 8-byte values. In either case, it's also possible to push and pop 2-byte values, but as far as I know you'd never want to do that.)

The stack isn't just an undifferentiated chunk of memory; it's divided into sections called *stack frames*. Whenever a function is called, it allocates some memory at the top of the stack by decreasing the stack pointer. This memory is the function's stack frame. Just before the function returns, it deallocates its stack frame, restoring the stack pointer to its previous value. We'll store the base of the current stack frame in the RBP register, which is the usual approach. We can refer to data in the current stack frame relative to the address stored in RBP. That way we don't need absolute addresses, which we can't know in advance. Since the stack grows toward lower memory addresses, any address in the current stack frame will be lower than the address stored in RBP, which is why we refer to local variables with operands like `-4(%rbp)`. We can also refer to data in the caller's stack frame, like function arguments, relative to RBP. We'll need to do that later when we implement function calls. (Alternatively, we could refer to local variables and parameters relative to RSP, and not bother with RBP at all; some compilers do this as an optimization. We'll stick with RBP-relative addressing because the resulting assembly is easier to understand.)

So, the first thing a function needs to do is set up a new stack frame, and the last thing it needs to do before it returns is restore the caller's stack frame. The function prologue sets up the stack frame in three instructions, as shown in Figure 3-3:

1. `pushq %rbp` saves the current value of RBP, the address of the base of the caller's stack frame, onto the stack. We save it because we'll need it to restore the caller's stack frame later. This value will be at the bottom of the new stack frame established by the next instruction.

2. `movq %rsp, %rbp` makes the top of the stack the base of the new

stack frame. At this point, the top and bottom of the current stack frame are the same. The current stack frame holds exactly one value, which both RSP and RBP point to: the base of the caller's stack frame, which we saved in the previous instruction.

3. `subq $n, %rsp` decrements the stack pointer by `n` bytes. The stack frame now has `n` bytes available to store local and temporary variables. In Figure 3-3, this instruction allocates 24 bytes, enough space for six 4-byte integers. It would also work to just push values onto the stack as needed, instead of allocating space for all of them up front, but most compilers don't do that. One problem with that approach is that you can only push 8-byte values. That's inconvenient when you want to store 4-byte integers, like we do right now.

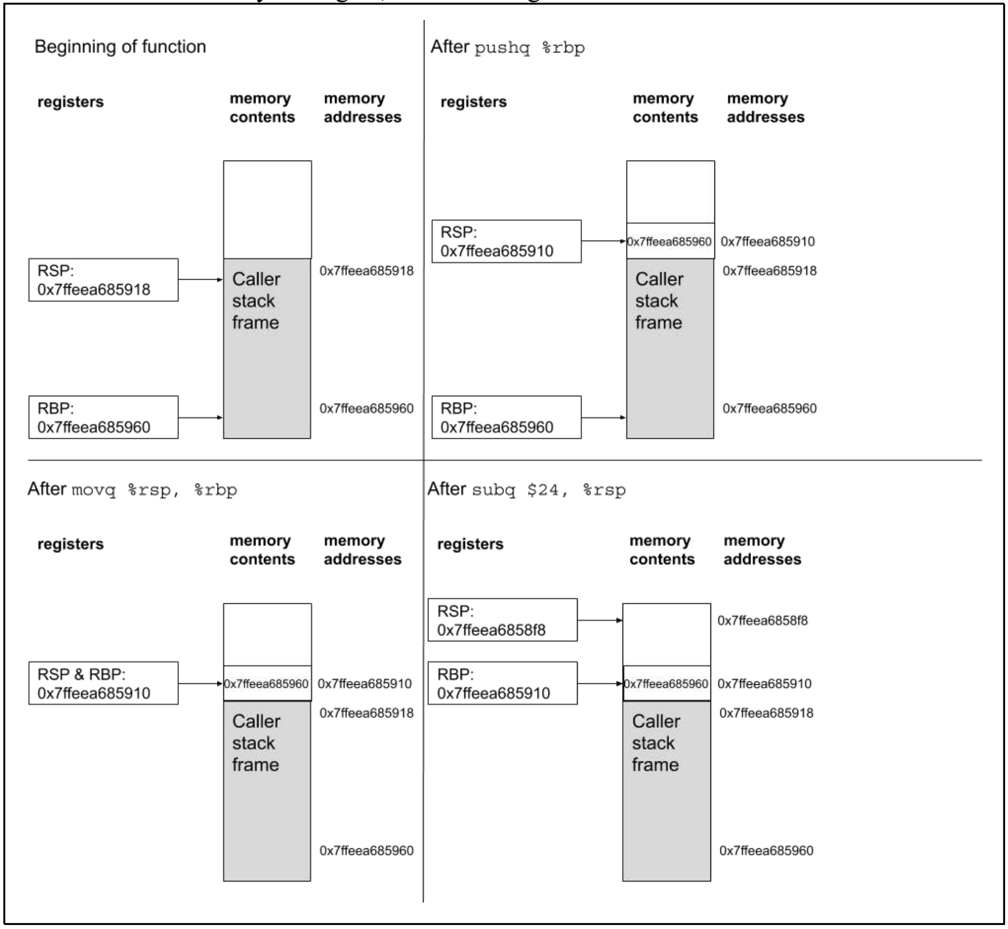

Figure 3-3 State of the stack at each point in the function prologue

The function epilogue needs to restore the caller's stack frame; that means RSP and RBP need to have the same values they did before the function prologue. This requires two instructions, as shown in Figure 3-4:

1. `movq %rbp, %rsp` puts us back where we were after the second instruction of the function prologue: both RSP and RBP point to bottom of the current stack frame, which holds the caller's value for RBP.

2. `popq %rbp` reverses the first instruction of the function prologue and restores the caller's values for both the RSP and RBP registers. It restores RBP because the value at the top of the stack was the base address of the caller's stack frame that we stored in the first instruction of the prologue. It restores RSP because it pops the last value in this stack frame off the stack, leaving RSP pointing to the top of the caller's stack frame.

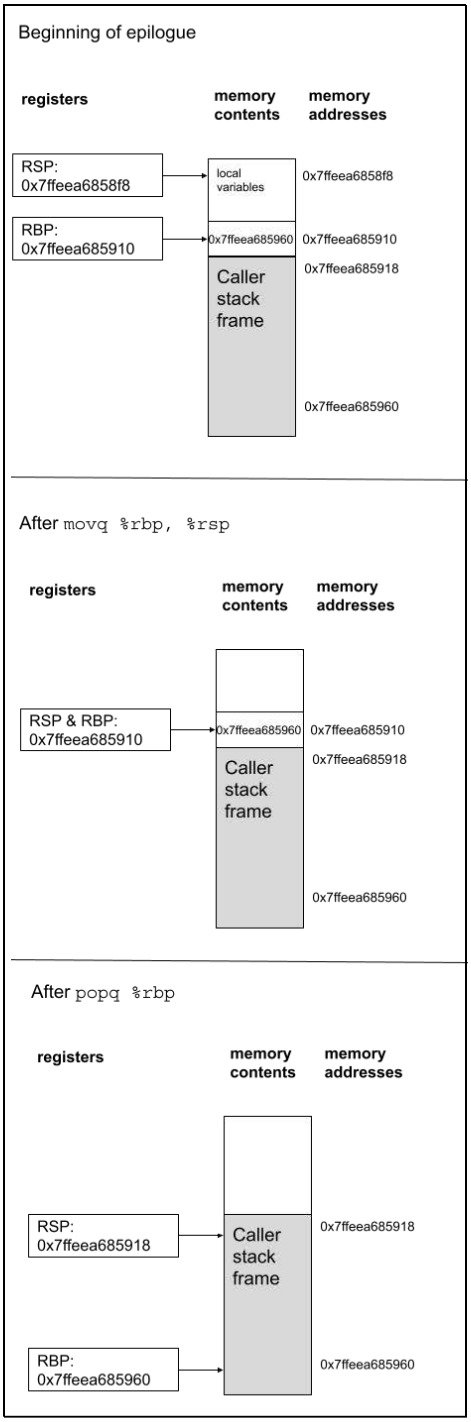

Figure 3-4 State of the stack at each point in the function epilogue

Now that we know what output our compiler should produce, we're ready to continue coding. Let's start by extending the lexer and parser.

Extending the Lexer

You'll need to extend the lexer to recognize three new tokens:

`~` : a tilde, the bitwise complement operator

`-` : a hyphen, the negation operator

`--` : two hyphens, the decrement operator

You won't implement the decrement operator in this chapter, but you still need to add a token for it. Otherwise, your compiler will accept programs it should reject, like this one:

```
int main() {
    return --2;
}
```

Listing 3-3 An invalid C program using the decrement operator

This shouldn't compile, because you can't decrement a constant. But if your compiler doesn't know that `--` is a distinct token, it will think Listing 3-3 is equivalent to:

```
int main() {
    return -(-2);
}
```

Listing 3-4 A valid C program with two negation operators in a row

which is a perfectly valid program. Your compiler should reject language features you haven't implemented—it shouldn't compile them incorrectly. That's why your lexer needs to know that `--` is a single token, not just two negation operators in a row. (On the other hand, the lexer should lex `~~` as two bitwise complement operators in a row. Expressions like `~~2` are perfectly valid.)

You can process the new tokens exactly the same way you handled punctuation like `;` and `(` in the previous chapter. First, you'll need to define a regular expression for each new token—the regular expressions here will just be the strings `~`, `-`, and `--`. Next, have your lexer check the input against these new regexes, as well as the regexes from the previous chapter, every time it tries to produce a token. Remember that when the start of the input stream matches more than one possible token, you should always choose the longest one. So, if your input stream ever starts with `--`, you'll parse it as a decrement operator rather than two negation operators.

Testing the Lexer

The lexer should successfully lex all the test cases for this chapter, including the valid test programs in *tests/chapter_3/valid* and the invalid test programs in *tests/chapter_3/invalid_parse*. To test your lexer against all the test cases so far, run:

```
$ ./test_compiler /path/to/your_compiler --chapter 3 --stage
lex
```

This will also run all the lexing test cases from Chapter 2, to make sure your lexer can still handle them.

Extending the Parser

To parse the new operators in this chapter, we first need to extend the AST and formal grammar we defined in Chapter 2. Let's look at the AST first. Since unary operations are expressions, we'll represent them with a new constructor for the `exp` AST node. Here's the updated AST definition, with new parts bolded:

```
program = Program(function_definition)
function_definition = Function(identifier name, statement
body)
statement = Return(exp)
exp = Constant(int) | Unary(unary_operator, exp)
unary_operator = Complement | Negate
```

Listing 3-5 Abstract syntax tree with unary operations

The updated rule for `exp` indicates that an expression can be either a constant integer or a unary operation. A unary operation consists of one of the two unary operators, `Complement` or `Negate`, applied to an inner expression. Notice that the definition of `exp` is recursive: the `Unary` constructor for an `exp` node contains another `exp` node. That lets us construct arbitrarily deeply nested expressions, like `-(~(-~-(-4)))`.

We also need to make the corresponding changes to the grammar:

```
<program> ::= <function>
<function> ::= "int" <identifier> "(" ")" "{" <statement>
"}"
<statement> ::= "return" <exp> ";"
<exp> ::= <int> | <unop> <exp> | "(" <exp> ")"
```

```
<unop> ::= "-" | "~"
<identifier> ::= ? An identifier token ?
<int> ::= ? A constant token ?
```

Listing 3-6 Formal grammar with unary operations

Listing 3-6 includes a new production rule for unary expressions, and a new `<unop>` symbol to represent the two unary operators. Those changes correspond exactly with the addition to the AST in Listing 3-5. We've also added a third production rule for the `exp` symbol, which doesn't correspond to anything in Listing 3-5. This rule just indicates that if you wrap an expression in parentheses, the result is still an expression. It doesn't have a corresponding constructor in the AST because the rest of the compiler doesn't need to distinguish between an expression wrapped in parentheses and the same expression without parentheses. The expressions `1`, `(1)`, and `((((1))))` should all be represented by the same AST node: `Constant(1)`.

The decrement operator `--`, doesn't show up anywhere in this grammar. That means your parser should fail if it encounters a `--` token.

To update the parsing stage, you first need to modify your compiler's AST data structure to match Listing 3-5. Then you need to update your recursive descent parsing code to reflect the changes in Listing 3-6. Parsing an expression is a bit more complicated than it was in the previous chapter, because the `<exp>` symbol has three different production rules and you need to figure out which one to apply. This pseudocode sketches out how to parse an expression:

```
  parse_exp(tokens):
1     next_token = peek(tokens)
2     if next_token is an int:
        --snip--
3     else if next_token is "~" or "-":
4         operator = parse_unop(tokens)
5         inner_exp = parse_exp(tokens)
6         return Unary(operator, inner_exp)
7     else if next_token == "(":
          take_token(tokens)
          inner_exp = parse_exp(tokens)
          expect(")", tokens)
8         return inner_exp
      else:
          fail()
```

Listing 3-7 Pseudocode for parsing an expression

The first step is looking at the next token in the input to figure out which production rule to apply. We call `peek` 1 to look at this token without removing it from the input stream. Once we know which production rule to use, we'll want to process the whole input, including that first token, using that rule. That's why we don't want to consume this token from the input just yet. (Like I mentioned in the last chapter, you might not consume tokens from the input stream exactly as I've described here, so you might look at the first token without actually calling a `peek` function.)

If the expression we're about to parse is valid, `next_token` should be an integer, a unary operator, or an open parenthesis. If it's an integer 2, we can parse it exactly the same way we did in the previous chapter. If it's a unary operator 3, we need to apply the second production rule for `<exp>` from Listing 3-6 to construct a unary expression. This rule is `<unop> <exp>`, so we'll parse the unary operator and then the inner expression. The `<unop>` symbol is a single token, `next_token`, which we've already inspected. In Listing 3-7, we handle `<unop>` in a separate function 4 (`parse_unop`, whose definition I've omitted). In practice, it might be unnecessary to define a separate function to parse just one token. Either way, we'll end up with a very simple AST node representing the appropriate unary operator. The operator should be followed by an `<exp>` symbol, which we'll process with a recursive call to `parse_exp` 5. (This is the recursive part of "recursive descent.") That call should return an `exp` AST node representing the operand of the unary expression. Now we have AST nodes for both the operator and the operand, so we can return the AST node for the whole unary expression 6.

If `next_token` is an open parenthesis 7, it should be immediately followed by a valid expression, so we remove the parenthesis from the input stream and call `parse_exp` recursively to handle the expression that follows. The inner expression should be followed by a closing parenthesis to balance out the opening parenthesis we already processed. We call `expect` to remove that closing parenthesis or throw a syntax error if it's missing. Since the AST doesn't need to indicate that there were parentheses, we can just return the inner expression as-is 8.

If `next_token` isn't an integer, a unary operator, or an open parenthesis, the expression must be malformed, so we throw a syntax error.

Testing the Parser

The parser should be able to handle every valid test case in *tests/chapter_3/valid*, and raise an error on every invalid test case in *tests/chapter_3/invalid_parse*. It should also continue to handle valid and invalid test cases from the last chapter correctly. To test your parser against the test cases from this chapter and last chapter, run:

```
$ ./test_compiler /path/to/your_compiler --chapter 3 --stage
parse
```

TACKY: A New Intermediate Representation

Converting the AST to assembly isn't as straightforward as it was in the last chapter. C expressions can have nested sub-expressions, and assembly instructions can't. A single expression like `-(~2)` needs to be broken up into two assembly instructions: one to apply the inner bitwise complement operation, and one to apply the outer negation operation.

We'll bridge the gap between C and assembly using a new intermediate representation (IR), *three-address code (TAC)*. In three-address code, the operands of each instruction must be constants or variables, not nested expressions. That means each instruction uses at most three values: two operands and a destination. (It would be more accurate to call this two-address code until we implement binary operators in the next chapter.) To rewrite nested expressions in three-address code, we often need to introduce new temporary variables. For example, `return 1 + 2 * 3` would become:

```
tmp0 = 2 * 3
tmp1 = 1 + tmp0
return tmp1
```

Listing 3-8 Three-address code for `return 1 + 2 * 3`

There are two main reasons to use three-address code instead of converting an AST directly to assembly. The first is that it lets us handle major structural transformations, like removing nested expressions, separately from the details of assembly language, like figuring out which operands are valid for which instructions. This lets us keep each compiler pass small, instead of having one really big compiler pass handling all those concerns. The second reason is that three-address code is well-suited to several of the optimizations we'll implement in Part III. It has a simple, uniform structure, which makes it easy to answer questions like "is the result of this expression ever used?" or "will this variable always the have same value?" The answers to those questions will determine what optimizations are safe to perform.

MULTIPLE LANGUAGES, MULTIPLE TARGETS

Intermediate representations like three-address code are useful for one other reason, although it isn't relevant to this project. An intermediate representation can provide a common target for multiple source languages and a common starting point for assembly generation for multiple target architectures. The LLVM compiler framework is a great example of this: it supports several frontends and backends using a single intermediate representation. If you want to compile a new programming language, you can just compile it to the LLVM IR, and then LLVM can do all the work of optimizing that IR and producing machine code for a bunch of different CPU architectures. Or, if you want to run software on some exotic new CPU architecture, you can just write a backend that converts the LLVM IR into machine code for that architecture, and you'll automatically be able to compile any language with an LLVM frontend for that architecture.

It's pretty normal for compilers to use some sort of three-address code internally, but the details vary. I've decided to name the intermediate representation in this book *TACKY*. (Naming your intermediate representations is, in my opinion, one of the best parts of compiler design.) I made up TACKY for this book, but it's similar to three-address code in other compilers.

Defining TACKY

We can define TACKY in ASDL, just like our other intermediate representations. It looks almost, but not quite, like the AST definition from Listing 3-4:

```
program = Program(function_definition)
function_definition = Function(identifier, 1instruction*
body)
instruction = Return(val) | Unary(unary_operator, val src,
val dst)
val = Constant(int) | Var(identifier)
unary_operator = Complement | Negate
```

Listing 3-9 Definition of TACKY

In TACKY, a function body consists of a list of instructions1, not just a single statement. In this respect, it's similar to the assembly AST we defined in the previous chapter. For now, TACKY has two instructions. `Return` returns a value. `Unary` performs some unary operation on `src`, the source value for the expression, and stores the result in `dst`, the destination. Both of these instructions operate on `val`s, which can be either constant integers (`Constant`) or temporary variables (`Var`).

TACKY makes a couple of assumptions that aren't explicit in Listing 3-9. The first assumption is that the `dst` of a unary operation will be a temporary `Var`, not a `Constant`. Trying to assign a value to a constant wouldn't make sense. The second assumption is that you'll always assign a value to a temporary before you use it. Right now, the only way to assign a value to a variable is by making it the `dst` of a unary operation. There are two ways to use a variable: by returning it, or by using it as the `src` of a unary operation. Because we're generating TACKY from an AST that we know is valid, we can guarantee that both of those assumptions hold.

You'll need to define a data structure for TACKY, just like you did for the AST and assembly AST. It can be similar to the data structures you used for the AST and assembly AST. For example, if you defined a separate algebraic datatype or abstract class for each node in the assembly AST, you'll want to take the same approach here. Once you have your data structure, you're ready to write the IR generation stage, which converts the AST from Listing 3-5 into TACKY.

Generating TACKY

Your IR generation pass needs to take an AST in the form defined in Listing 3-5, and return a TACKY AST in the form defined in Listing 3-9. The tricky part is turning an expression into a list of instructions; once you have that figured out, handling all the other AST nodes is easy. Table 3-1 lists a few examples of ASTs and the resulting TACKY:

Table 3-1 TACKY representations of unary expressions

AST	**TACKY**
`Return(Constant(3))`	`Return(Constant(3))`

`Return(Unary(Complement, Constant(2)))`	`Unary(Complement, Constant(2), Var(tmp0))`
	`Return(Var(tmp0))`

`Return(Unary(Negate,`
`Unary(Complement,`
`Unary(Negate,`
`Constant(8))))`
	`Unary(Negate, Constant(8), Var(tmp0))`
	`Unary(Complement, Var(tmp0), Var(tmp1))`
	`Unary(Negate, Var(tmp1), Var(tmp2))`
	`Return(Var(tmp2))`

In the examples above, we convert each unary operation into a `Unary` TACKY instruction, starting with the innermost expression and working our way out. We store the result of each `Unary` instruction in a temporary variable, which we then use in the outer expression or return statement. The pseudocode in Listing 3-10 describes how to generate these TACKY instructions.

```
emit_tacky(e, instructions):
1   match e with
    | 2 Constant(c) -> return 3 Constant(c)
     | Unary(op, inner) ->
4        src = emit_tacky(inner, instructions)
5       dst_name = make_temporary()
        dst = Var(dst_name)
        tacky_op = convert_unop(op)
6       instructions.append(Unary(tacky_op, src, dst))
        return dst
```

Listing 3-10 Pseudocode to convert an expression into a list of TACKY instructions

This pseudocode emits the instructions needed to calculate an expression by appending them to the `instructions` argument. It also returns a TACKY `val` that represents the result of the expression, which we'll use when we translate the outer expression or statement.

The `match` statement in Listing 3-10 checks which type of expression we're translating, then runs the clause to handle that kind of expression 1. If the expression is a constant, we'll just return the equivalent TACKY `Constant` without generating any new instructions. Note that the `Constant` constructs at 2 and 3 are different; 2 is a node in the original AST, while 3 is a node in the TACKY AST. (The same is true for the two `Unary` constructs that appear in the next clause.)

If `e` is a unary expression, we'll construct TACKY values for the source and destination. First, we'll call `emit_tacky` recursively on the source expression to get the corresponding TACKY value 4. This will also generate the TACKY instructions to calculate that value. Then, we'll create a new temporary variable for the destination 5. The `make_temporary` helper function will generate a unique name for this variable. We'll use another helper function, `convert_unop`, to convert the unary operator to its TACKY equivalent. I won't provide pseudocode for either of these helper functions, since they're very simple. Once we have our source, destination, and unary operator, we'll construct the `Unary` TACKY instruction and append it to the `instructions` list 6. Finally, we'll return `dst` as the result of the whole expression.

Testing the TACKY Generator

The TACKY generator should be able to handle every valid test case from this chapter and the one before. To test this stage, we'll run the whole compiler and check whether it succeeds or fails, without inspecting its output. You can run those tests with:

```
$ ./test_compiler /path/to/your_compiler --chapter 3 --stage
codegen
```

The TACKY stage shouldn't encounter any invalid test cases, because the lexer and parser should catch them first.

Implementation Tips

Generate globally unique names. In the TACKY examples in Table 3-1, it's clear that giving two temporary variables the same name would be an error. In later chapters, we'll want to guarantee that no two temporaries share the same name, even if they're in different functions. That makes IR generation easier, because you don't have to think about whether it's safe for two temporaries to have the same name or not. So, you'll want a convenient way to generate unique names. One easy solution is to maintain a global counter; to generate a unique name, just increment the counter and use its new value (or its new value plus some descriptive string) as your variable name. Because these names won't appear in assembly, they don't need to follow any particular naming convention; they just have to be unique.

Handle expressions in general, not return statements in particular. Right now, expressions only appear in return statements, but they'll show up in other kinds of statements in later chapters. Make sure your solution can be extended to handle expressions that aren't in return statements.

Assembly Generation

TACKY is closer to assembly, but it still doesn't specify exactly which assembly instructions we need. The next step is converting the program from TACKY into the assembly AST we defined in the last chapter. We'll do this in three small compiler passes. First, we'll produce an assembly AST, but still refer to temporary variables directly. Next, we'll replace those variables with concrete addresses on the stack. That step will result in some invalid instructions, because many x64 assembly instructions can't use memory addresses for both operands. So, in the last compiler pass, we'll rewrite the assembly AST to fix any invalid instructions.

Converting TACKY to Assembly

First, we need to extend the assembly AST we defined in the last chapter. We need a way to represent the `neg` and `not` instructions that we used in Listing 3-2. We also need to decide how, or whether, we'll represent the function prologue and epilogue in the assembly AST.

We have a few different options for handling the prologue and epilogue. We could go ahead and add the push, pop, and sub instructions to the assembly AST. We could add high-level instructions that correspond to the entire prologue and epilogue, instead of maintaining a 1-1 correspondence between assembly AST constructs and assembly instructions. Or we could omit the function prologue and epilogue entirely, and add them during code emission. The assembly AST below just includes an instruction for decrementing the stack pointer (the third instruction in the function prologue) so we can record how many bytes we need to subtract. Because the rest of the prologue and epilogue are completely fixed, we can easily add them during code emission even if they're not included in the assembly AST. That said, the other approaches to representing the function prologue and epilogue can also work, so feel free to choose whichever seems best to you.

We'll also introduce *pseudoregisters* to represent temporary variables. We'll be able to use pseudoregisters exactly the same way as real registers in the assembly AST; the only difference is that they don't correspond to hardware registers, so we have an unlimited supply of them. Because they aren't real registers, they can't appear in the final assembly program; they'll need to be replaced by real registers or memory addresses in a later compiler pass. For now, we'll assign every pseudoregister to its own address in memory. In Part III, we'll write a *register allocator*, which will speed up the program by assigning as many pseudoregisters as possible to hardware registers instead of memory.

Here's the updated assembly AST, with new parts bolded:

```
program = Program(function_definition)
function_definition = Function(identifier name, instruction*
instructions)
instruction = Mov(operand src, operand dst)
            | Unary(unary_operator, operand)
            | AllocateStack(int)
            | Ret
unary_operator = Neg | Not
operand = Imm(int) | Reg(reg) | Pseudo(identifier) |
Stack(int)
reg = AX | R10
```

Listing 3-11 Assembly definition with unary operators

The instruction node has a couple of new constructors to represent our new assembly instructions. We'll represent both new unary instructions with the Unary constructor. Since this constructor represents a single not or `neg` instruction, it takes just one operand that's used as both source and destination. The `AllocateStack` constructor represents the third instruction in the function prologue, `subq $n, %rsp`. Its one child, an integer, indicates the number of bytes we'll subtract from RSP.

We also have several new instruction operands. The `Reg` operand can represent either of the two hardware registers we've seen so far: EAX and R10D. The `Pseudo` operand lets us use an arbitrary identifier as a pseudoregister. We'll use this to refer to the temporary variables we produced while generating TACKY. Ultimately, we need to replace every pseudoregister with a location on the stack; we'll represent those with the `Stack` operand, which indicates the stack address at the given offset from RBP. For example, in Listing 3-2 we used `-4(%rbp)` as an operand. We'd represent this as `Stack(-4)` in the assembly AST.

NOTE Every hardware register has several aliases, depending on how many bytes of the register you need. EAX refers to the lower 32 bits of the 64-bit RAX register, and R10D refers to the lower 32 bits of the 64-bit R10 register. The names AX and R10B refer to the lower 8 bits of RAX and R10, respectively. Register names in the assembly AST are size-agnostic, so `AX` in Listing 3-11 can refer to the register alias RAX, EAX, or AX, depending on context.

Now we can write a straightforward conversion from TACKY to assembly, given in Table 2-2 below:

Table 3-2 Conversion from TACKY to Assembly

TACKY	**Assembly**
Top-level constructs	
`Program(function_definition)`	`Program(function_definition)`
`Function(name, instructions)`	`Function(name, instructions)`
Instructions	

`Return(val)`	`Mov(val, Reg(AX))`
	`Ret`

`Unary(unary_operator, src, dst)`	`Mov(src, dst)`
	`Unary(unary_operator, dst)`

Operators	
`Complement`	`Not`

`Negate`	`Neg`
Operands	
`Constant(int)`	`Imm(int)`
`Var(identifier)`	`Pseudo(identifier)`

Since our new assembly instructions use the same operation for the source and destination, we just copy the source value into the destination before issuing the `neg` or `not` instruction. Note that we're not using the `AllocateStack` instruction yet; we'll add it in the very last stage before code emission, once we know how many bytes we need to allocate. We're also not using any `Stack` operands; we'll replace every `Pseudo` operand with a `Stack` operand in the next compiler pass. And we're not using the R10D register; we'll introduce it when we rewrite invalid instructions.

Replacing Pseudoregisters

Next, we'll write a compiler pass to replace each `Pseudo` operand with a `Stack` operand, leaving the rest of the assembly AST unchanged. In Listing 3-2, we used two stack locations: `-4(%rbp)` and `-8(%rbp)`. We'll stick with that pattern: the first temporary variable we assign a value to will be at `Stack(-4)`, the next will be at `Stack(-8)`, and so on. We'll subtract four for each new variable, since every temporary variable is a 4-byte integer. You'll need to maintain a map from identifiers to offsets as you go, so you can replace each pseudoregister with the same address on the stack every time it appears. For example, if you were processing the following list of instructions:

```
Mov(Imm(2), Pseudo(A))
Unary(Neg, Pseudo(A))
```

you would need to make sure that `Pseudo(A)` was replaced with the same `Stack` operand in both instructions.

This compiler pass should also return the stack offset of the final temporary variable, because that tells us how many bytes to allocate on the stack in the final compiler pass.

Fixing Up Instructions

Now we need to traverse the assembly AST one more time and make two small fixes. The first fix is inserting the `AllocateStack` instruction at the very beginning of the instruction list in the `function_definition`. The integer argument to `AllocateStack` should be the stack offset of the last temporary variable we allocated in the previous compiler pass. That way, we'll allocate enough space on the stack to accommodate every address we use. For example, if we replace three temporary variables, replacing the last one with `-12(%rbp)`, we'll insert `AllocateStack(12)` at the front of the instruction list.

The second fix is rewriting invalid `Mov` instructions. When we replaced a bunch of pseudoregisters with stack addresses, we may have ended up with `Mov` instructions where both the source and destination are `Stack` operands. In particular, this will happen if the unary expression in your program has at least one level of nesting. But `mov`, like many other instructions, can't have memory addresses in both the source and the destination. If you try to assemble a program with an instruction like `movl -4(%rbp), -8(%rbp)`, the assembler will reject it. Whenever you encounter an invalid `mov` instruction, you'll need to rewrite it to first copy from the source address into R10D, and then copy from R10D to the destination. For example,

```
movl -4(%rbp), -8(%rbp)
```

would become

```
movl -4(%rbp), %r10d
movl %r10d, -8(%rbp)
```

I've chosen R10D as a scratch register because it doesn't serve any other special purpose. Some registers are used by particular instructions; for example, the `idiv` instruction, which performs division, requires the dividend to be stored in EAX. Other registers are used to pass arguments during function calls. Using any of these registers for scratch at this stage could cause conflicts later. For example, you might copy a function argument into the correct register, but then accidentally overwrite it while using that register to transfer a different value between memory addresses. But because R10D doesn't have any special purpose, we don't have to worry about that kind of conflict.

Testing Code Generation

Once you've implemented the assembly generation passes, you can test them exactly the same way as the TACKY generator:

```
$ ./test_compiler /path/to/your_compiler --chapter 3 --stage
codegen
```

Implementation Tips

Plan ahead for Part II. The `Unary` instruction, like `Mov`, will eventually need to record type information; consider defining it in a way that will make it easier to add type information later on.

Define a register datatype. It might seem easiest to store register names as strings, but I think you're better off defining a new datatype to represent them. Like I mentioned earlier, registers in our assembly AST are size-agnostic, but register names in the final assembly program are not. Your code will be clearer if you distinguish between registers in the assembly AST, which aren't yet associated with a particular integer size, and the registers names that will appear in the final assembly program.

Extending the Code Emitter

Finally, we need to extend the code emission stage to handle our new constructs and print out the function prologue and epilogue. Here's how to print out each construct, with new and changed constructs bolded:

Table 3-3 Formatting assembly

Assembly Construct	**Output**
Top-level constructs	
Program(function_definition)	(just print out the function definition)

Function(name, instructions)	`    .globl <name>`
	`<name>:`
	`    pushq    %rbp`
	`    movq     %rsp, %rbp`
	`    <instructions>`

Instructions	
Mov(src, dst)	`movl <src>, <dst>`

Ret	`    movq     %rbp, %rsp`
	`    popq     %rbp`
	`    ret`

Unary(unary_operator, operand)	`<unary_operator> <operand>`

AllocateStack(int)		`subq $<int>, %rsp`
Operators		
Neg		`negl`
Not		`notl`
Operands		
Reg(AX)		`%eax`
Reg(R10)		`%r10d`
Stack(int)		`<int>(%rbp)`
Imm(int)		`$<int>`

We'll always insert the function prologue right after the function's label. We'll also emit the whole function epilogue whenever we encounter a single `ret` instruction. Because RBP and RSP contain memory addresses, which are eight bytes, we'll operate on them using quadword instructions, which have a `q` prefix. Note that the program now includes two versions of the `mov` instruction: `movl` and `movq`. They're identical apart from the size of their operands.

Testing the Whole Compiler

Once you've updated the code emission stage, your compiler should produce correct assembly for all the test cases in this chapter. To test it out, run:

```
$ ./test_compiler /path/to/your_compiler --chapter 3
```

Just like in the previous chapter, this will compile all the valid examples, run them, and verify the return code. It also runs the invalid examples, but those should already fail at the parsing stage.

Summary

In this chapter, you extended your compiler to implement negation and bitwise complement. You also implemented a new intermediate representation, wrote a couple different compiler passes that transform assembly code, and learned how stack frames are structured. Next, you'll implement binary operations like addition and subtraction. The changes to the backend in the next chapter will be pretty simple; the tricky part will be getting the parser to respect operator precedence and associativity.

4 BINARY OPERATORS

In this chapter, you’ll implement five new operators: addition, subtraction, multiplication, division, and the modulo operator. These are all *binary operators*, which take two operands. This chapter won’t require any new compiler stages; you’ll just need to extend each of the stages you’ve already written. In the parsing stage, we’ll see why recursive descent parsing doesn’t work well for binary operators. You’ll learn about a different technique, *precedence climbing*, that will be easier to build on in later chapters. Precedence climbing is the last major parsing technique we'll need. Once it's in place, we'll be able to add new syntax with relatively little effort for the rest of the book. In the code generation stage, we'll introduce several new assembly instructions that perform binary operations. As usual, we'll start with the lexer.

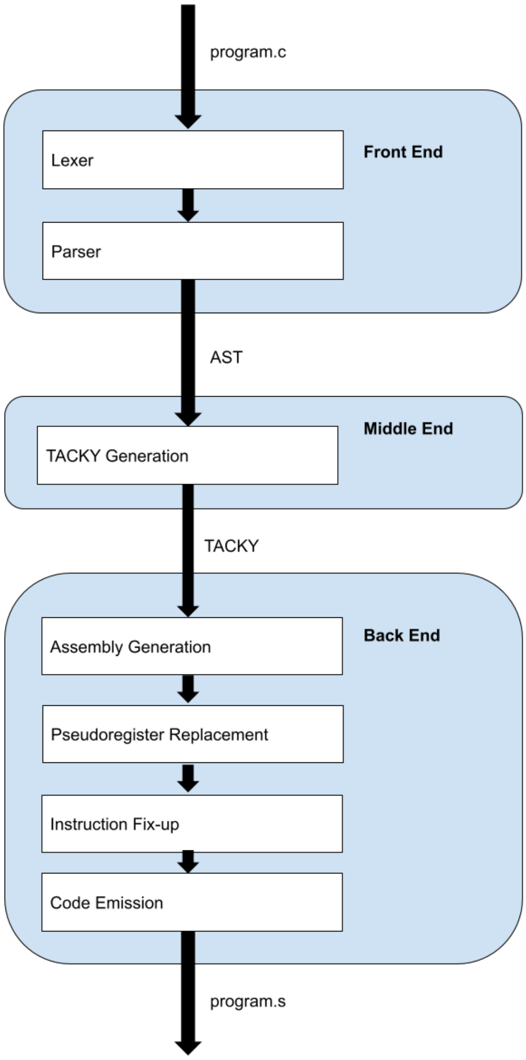

Figure 4-1 Stages of the compiler

Extending the Lexer

The lexer will need to recognize four new tokens:

`+` : a plus sign, the operator for addition

`*` : an asterisk, the operator for multiplication

`/` : a forward slash, the division operator

`%` : a percent sign, the modulo operator

This list doesn't include the `-` token, because you already added it in the last chapter. The lexing stage doesn't distinguish between negation and subtraction; it should produce the same token either way.

You can implement these tokens the same way you did the single-character tokens in earlier chapters.

Testing the Lexer

You know the drill. Your lexer shouldn't fail on any of the test cases in this chapter.

```
$ ./test_compiler /path/to/your_compiler --chapter 4 --stage
lex
```

Extending the Parser

In this chapter, we'll need to add another kind of expression to the AST: binary operations. Listing 4-1 gives the updated AST definition:

```
program = Program(function_definition)
function_definition = Function(identifier name, statement
body)
statement = Return(exp)
exp = Constant(int)
    | Unary(unary_operator, exp)
    | Binary(binary_operator, exp, exp)
unary_operator = Complement | Negate
binary_operator = Add | Subtract | Multiply | Divide | Mod
```

Listing 4-1 Abstract syntax tree with binary operations

There are a couple things to note about this AST definition. The first is that the parser, unlike the lexer, distinguishes between negation and subtraction. A `-` token will be parsed as either `Negate` or `Subtract`, depending on where it appears in an expression.

The second point is that the structure of the AST determines the order in which we evaluate nested expressions. Let's look at a couple examples to see how the AST's structure controls the order of operations. The AST in Figure 4-2 represents the expression `1 + (2 * 3)`, which evaluates to `7`.

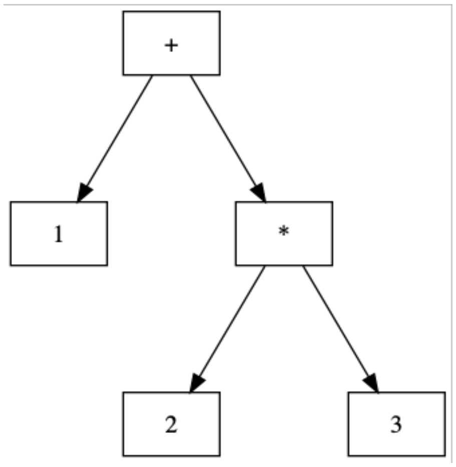

Figure 4-2 AST for `1 + (2 * 3)`

The `+` operation has two operands: `1` and `(2 * 3)`. If you were going to evaluate this expression, you would need to calculate `2 * 3` first, and then add `1` to the result. The AST in Figure 4-3, on the other hand, represents the expression `(1 + 2) * 3`, which evaluates to `9`:

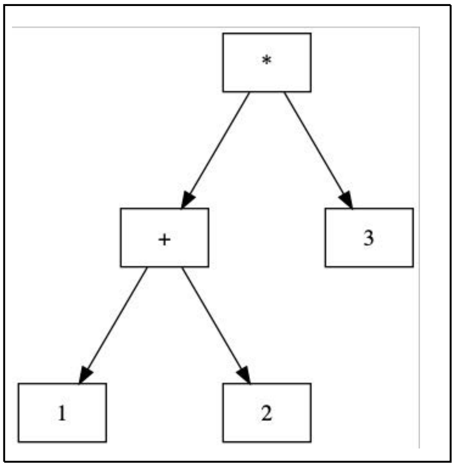

Figure 4-3 AST for (1 + 2) * 3

In this case, you would need to evaluate `1 + 2` first, then multiply by

3. As a general rule, before evaluating an AST node you need to evaluate both of its children. This pattern, where you need to process a node's children before you process the node itself, is called *post-order traversal*. (Note that any tree data structure can be traversed in post-order, not just ASTs.)

Your compiler will traverse the AST to generate code, not to evaluate expressions, but the idea is the same. When you convert the AST for a binary expression to TACKY, you need to generate instructions to calculate both operands, then generate instructions for the operator itself. (We also used post-order traversal to process unary operations in the last chapter.)

The point of all this is that it's very important for your parser to group nested expressions correctly, because if you try to parse `1 + (2 * 3)` but end up with the AST from Figure 4-3, you'll end up compiling the program incorrectly.

The examples we just looked at used parentheses to explicitly group nested expressions. Some expressions, like `1 + 2 * 3`, don't parenthesize every nested expression. In those cases, we group expressions based on the *precedence* and *associativity* of the operators. Operators with higher precedence are evaluated first; since `*` has higher precedence than `+`, you'd parse `1 + 2 * 3` as `1 + (2 * 3)`. Associativity tells you how to handle operators at the same precedence level. If an operation is *left-associative*, you apply the operator on the left first, and if it's *right-associative*, you apply the operator on the right first. For example, since addition and subtraction are left-associative, `1 + 2 - 3` would be parsed as `(1 + 2) - 3`. All the new operators in this chapter are left-associative, and there are two precedence levels: `*`, `/`, and `%` have higher precedence, and `+` and `-` have lower precedence.

The Trouble with Recursive Descent Parsing

It's surprisingly tricky to write a recursive descent parser that handles operator precedence and associativity correctly. To see why, let's try extending the grammar rule for expressions. The obvious rule would look something like this:

```
<exp> ::= 1  <int> | <unop> <exp> | "(" <exp> ")" | 2  <exp>
<binop> <exp>
```

Listing 4-2 A deceptively simple but unworkable grammar rule

The rule in Listing 4-2 corresponds to the AST definition: we added a new constructor to the AST, so it stands to reason that we can just add another production rule 2 to the grammar. But this production rule won't work with the recursive-descent parsing algorithm we've used up to now. If we try to use it, we'll run into two problems.

The first problem is that rule 2 is *ambiguous*: sometimes it allows you to parse a list of tokens in more than one way. Based on this rule, Figures 4-2 and 4-3 are equally valid parses of `1 + 2 * 3`. We need to know about the relative precedence of `+` and `*` to decide which parse to use, but the rule in Listing 4-2 doesn't capture that information.

The second problem is that rule 2 is *left-recursive*. That means the left-most symbol in this production rule for `<exp>` is, itself, `<exp>`. You can't parse a left-recursive rule with a recursive descent parser. First of all, it's impossible to decide which production rule to apply. Let's say your input starts with an `<int>` token. Maybe your expression is a single `<int>`, so you should apply production rule 1. Or maybe it's a more complex expression and the `<int>` is just the first operand of the first sub-expression, so you should apply production rule 2. There's no way to tell until you've parsed some of the input.

Even if you could determine which production rule to use, processing rule 2 would lead to unbounded recursion. The first symbol in this rule is an `<exp>`, so `parse_exp` would have to process that symbol by calling itself recursively. But, because `parse_exp` would be calling itself with exactly the same input, since it didn't consume any tokens before the recursive call, it would never terminate.

There are a couple of ways to solve these two problems. If you want a pure recursive descent parser, you can refactor the grammar to remove the ambiguity and left-recursion. Since that approach has some drawbacks, we'll use an alternative to recursive descent parsing called precedence climbing. However, it's helpful to take a look at the pure recursive-descent solution first.

The Adequate Solution: Refactoring the Grammar

If we refactor the grammar, we'll end up with one grammar rule for each precedence level:

```
<exp> ::= <term> { ("+" | "-") <term> }
<term> ::= <factor> { ("*" | "/" | "%") <factor> }
<factor> ::= <int> | <unop> <factor> | "(" <exp> ")"
```

Listing 4-3 A recursive descent-friendly grammar for binary operations

Using the grammar in Listing 4-3, there's only one way to parse `1 + 2 * 3`, and there's no left recursion. The curly braces indicate repetition, so a single `<exp>`, for example, can contain any number of `<term>`s. It might be a `<term>`, or `<term> + <term>`, or `<term> - <term> + <term>`, and so on. The parser then groups that long string of terms or factors into a left-associative tree. (Note that we can't use a rule like `<exp> ::= <term> "+" <exp>` because it would result in a right-associative tree.)

This approach works, but it gets more unwieldy as you add more precedence levels. We have three precedence levels now, if you count `<factor>`; we'll add four more when we introduce logical and relational operators in the next chapter. If we went with this approach, we'd need to add a new symbol to the grammar—and a corresponding function to our parser—for each precedence level we add. That's a lot of boilerplate, since the functions to parse all the different binary expressions will be almost identical.

The Better Solution: Precedence Climbing

Precedence climbing is a simpler way to parse binary expressions; it can handle production rules like `<exp> <binop> <exp>`. The basic idea is that every operator will have a numeric precedence level, and `parse_exp` will take a minimum precedence level as an argument. That lets you specify the appropriate precedence level for whatever sub-expression you're parsing. For example, let's say you just saw a `+` token, and now want to parse what comes next as the right-hand side of an addition expression—you would specify that it should only include operations that are higher-precedence than `+`. This solution makes it easy to add new operators; you have to assign those operators an appropriate numeric precedence level, but otherwise your parsing code doesn't need to change.

Mixing Precedence Climbing with Recursive Descent

Luckily, we can use precedence climbing here without rewriting the recursive descent parsing code we wrote earlier. We'll write a hybrid parser that uses precedence climbing for binary expressions, and recursive descent for everything else. Remember that in a recursive descent parser, we define one parsing function to handle each symbol. That makes it straightforward to mix the two approaches: we can just use precedence climbing in the `parse_exp` function, and recursive descent in the functions that parse all the other symbols. The `parse_exp` function will remove tokens from the input stream and return an `exp` AST node, just like a recursive descent-based parsing function would. But it will use a different strategy to get that result.

Since we already know how to parse unary and parenthesized expressions with recursive descent, let's represent those with a separate symbol from binary operations. That will make it easier to parse the two types of expressions using different techniques. Here's the resulting grammar:

```
<program> ::= <function>
<function> ::= "int" <identifier> "(" ")" "{" <statement>
"}"
<statement> ::= "return" <exp> ";"
<exp> ::= <factor> | <exp> <binop> <exp>
<factor> ::= <int> | <unop> <factor> | "(" <exp> ")"
<unop> ::= "-" | "~"
<binop> ::= "-" | "+" | "*" | "/" | "%"
<identifier> ::= ? An identifier token ?
<int> ::= ? A constant token ?
```

Listing 4-4 The final grammar to handle binary operations

A `<factor>` (which we were calling an `<exp>` in the last chapter) can be parsed with the usual recursive descent approach. (We'll keep calling this symbol a "factor," like we do in Listing 4-3, since it can appear as a term in a multiplication, division, or modulo expression.) It looks almost exactly like last chapter's rule for `<exp>`, except that we now allow binary operations as well as factors inside parentheses. That means `(1 + 2)` is a factor, because `"(" <exp> ")"` is a production rule for `<factor>`. However, `-1 + 2` is not, because `"-" <exp>` is not a production rule for `<factor>`. Because the rules for `<exp>` and `<factor>` refer to each other, the functions to parse those symbols will be mutually recursive. An `<exp>` is either a binary operation, defined in the obvious way, or it's just a factor.

The pseudocode to parse factors also looks almost the same as last chapter:

```
parse_factor(tokens):
    next_token = peek(tokens)
    if next_token is an int:
        --snip--
    else if next_token is "~" or "-":
        operator = parse_unop(tokens)
1□      inner_exp = parse_factor(tokens)
        return Unary(operator, inner_exp)
    else if next_token == "(":
        take_token(tokens)
2□      inner_exp = parse_exp(tokens)
        expect(")", tokens)
        return inner_exp
    else:
        fail()
```

Listing 4-5 Pseudocode for parsing a factor

The only difference is that we call parse_factor where we expect a <factor> 1, and parse_exp where we expect an <exp> 2; before, we just called parse_exp in both places.

Making Operators Left-Associative

Next, we need to figure out what parse_exp looks like. First, let's make the problem simpler by only considering the + and - operators, which are both at the same precedence level. To handle these operators, parse_exp needs to group expressions in a left-associative way, but it doesn't need to handle multiple precedence levels yet.

In this simple case, we'll encounter inputs like factor1 + factor2 - factor3 + factor4. These should always be parsed in a left-associative way to produce expressions like ((factor1 + factor2) - factor3) + factor4. As a result, the right operand of every expression, including sub-expressions, will be a single factor. For example, the right operand of (factor1 + factor2) is factor2, and the right operator of ((factor1 + factor2) - factor3) is factor3.

Once we realize that the right operand of an expression is always a single factor, we can write pseudocode to parse these expressions:

```
   parse_exp(tokens):
1      left = parse_factor(tokens)
       next_token = peek(tokens)
2      while next_token is "+" or "-":
           operator = parse_binop(tokens)
3          right = parse_factor(tokens)
4          left = Binary(operator, left, right)
           next_token = peek(tokens)
       return left
```

Listing 4-6 Parsing left-associative expressions without considering precedence level

In Listing 4-6, we start by parsing a single factor 1. This factor will be either the whole expression or the left operand of a larger expression. Then, we check if the next token is a binary operator 2. If it is, we consume it from the input and convert it to an AST node. Then we construct a binary expression where the left operand is everything we've parsed so far and the right operand is the next factor 4, which we get by calling parse_factor 3. We repeat this process until we see a token other than + or - after a factor; that means there are no binary expressions left to construct, so we're done.

Dealing with Precedence

Listing 4-6 lets us parse left-associative binary operators, but it doesn't handle different precedence levels. Now let's extend it to handle `*`, `/`, and `%`. These operators are also left-associative, but they're at a higher precedence level than `+` and `-`.

Once we add these operators, the right operand of every expression can be either a single factor, or a sub-expression involving only the new, higher-precedence operators. For example, `1 + 2 * 3 + 4` would be parsed as `(1 + (2 * 3)) + 4.` The right operand of the whole expression is a single factor, 4. The right operand of the inner sub-expression, `1 + (2 * 3)`, is a product, `2 * 3`.

We can be even more precise. If the outermost expression is a `+` or `-` operation, its right operand will only contain factors, `*`, `/`, and `%`. But if the outermost expression is itself a `*`, `/`, or `%` operation, its right operand must be single factor.

To generalize: whenever we're parsing an expression of the form `e1 <op> e2`, all the operators in `e2` should be higher-precedence than `<op>`. We can achieve this by tweaking the code from Listing 4-6:

```
parse_exp(tokens, min_prec):
    left = parse_factor(tokens)
    next_token = peek(tokens)
    while next_token is a binary operator and
precedence(next_token) >= min_prec:
        operator = parse_binop(tokens)
        right = parse_exp(tokens, precedence(next_token) +
1)
        left = Binary(operator, left, right)
        next_token = peek(tokens)
    return left
```

Listing 4-7 Parsing left-associative operators with precedence climbing

This pseudocode is our entire precedence climbing algorithm. The `min_prec` argument lets us state that all operators in the sub-expression we're currently parsing need to exceed some precedence level. For example, we could include only operators that are higher-precedence than `+`. We enforce this by comparing the precedence of the current operator to `min_prec` at each iteration of the while loop; we exclude the operator and anything that follows it from the current expression if its precedence is too low. Then, when we process the right-hand side of an operation, we set the minimum precedence higher than the precedence of the current operator. This guarantees that higher-precedence operators will be evaluated first.

Since operators at the same precedence level as the current operator won't be included in the right-hand expression, the resulting AST will be left-associative.

When you're calling `parse_exp` from any other function (including from `parse_factor`, to handle parenthesized expressions), you'll start with a minimum precedence of zero, so the result includes operators at every precedence level.

The code in Listing 4-7 requires us to assign every binary operator a precedence value; the values I've assigned are listed in Table 4-1.

Table 4-1 Precedence Values of Binary Operators

Operator	**Precedence**
`*`	50
`/`	50
`%`	50
`+`	45
`-`	45

The exact precedence values don't matter, as long as higher-precedence operators have higher values. The numbers I've chosen here give us plenty of room to add new lower-precedence operators in the next chapter.

Precedence Climbing in Action

Let's walk through an example where we parse the following expression:

```
1 * 2 - 3 * (4 + 5)
```

We'll trace the execution of our precedence-climbing code (Listing 4-7) as it parses this expression. In each code snippet below, I've added a level of indentation inside each function call, to make it easier to track how deep we are in the call stack.

We'll start by calling `parse_exp` on the whole expression with a minimum precedence of zero:

```
parse_exp("1 * 2 - 3 * (4 + 5)", 0):
```

Inside parse_exp, we'll start by parsing the first factor:

```
left = parse_factor("1 * 2 - 3 * (4 + 5)")
     = Constant(1)
next_token = "*"
```

This first call to parse_factor will just parse the token 1, returning Constant(1). Next, we peek at the token that follows, which is *. This token is a binary operator, and its precedence is greater than zero, so we enter the while loop.

The first iteration of the loop looks like this:

```
// loop iteration #1
operator = parse_binop("* 2 - 3 * (4 + 5)")
         = "*"
right = parse_exp("2 - 3 * (4 + 5)", 51)
          left = parse_factor("2 - 3 * (4 + 5)")
               = Constant(2)
          next_token = "-"
          // precedence(next_token) < 51
      = Constant(2)
left = Binary(*, Constant(1), Constant(2))
next_token = "-"
```

Inside the loop, parse_binop consumes next_token from the input, which leaves 2 - 3 * (4 + 5). Next, we need to call parse_exp recursively to get the right-hand side of this product. Since the precedence of * is 50, the second argument to parse_exp will be 51. In the recursive call, we again get the next factor (2) and the token that follows it (-). The - token is a binary operator, but its precedence is only 45; it doesn't meet the minimum precedence of 51, so we don't enter the while loop. Instead, we return Constant(2).

Back in the outer call to parse_exp, we use Binary to construct the AST node for 1 * 2 from the values we've parsed so far. Then, we check the next token to see whether we have more sub-expressions to process. The next token is -; we peeked at it, but didn't remove it from the input, inside the recursive call to parse_exp. Because - is a binary operator, and it exceeds our minimum precedence of zero, we jump back to the beginning of the while loop to parse the next sub-expression:

```
// loop iteration #2
operator = parse_binop("- 3 * (4 + 5)")
         = "-"
right = parse_exp("3 * (4 + 5)", 46)
          left = parse_factor("3 * (4 + 5)")
               = Constant(3)
          next_token = "*"
          // loop iteration #1
```

```
                 operator = parse_binop("* (4 + 5)")
                        = "*"
               right = parse_exp("(4 + 5)", 51)
                         left = parse_factor("(4 + 5)")
                                 parse_exp("4 + 5", 0)
                                 --snip--
                                 = Binary(+, Constant(4),
Constant(5))
                             = Binary(+, Constant(4),
Constant(5))
                    = Binary(+, Constant(4), Constant(5))
               left = Binary(*, Constant(3), Binary(+,
Constant(4), Constant(5)))
          = Binary(*, Constant(3), Binary(+, Constant(4),
Constant(5)))
  left = Binary(-,
                    Binary(*, Constant(1), Constant(2)),
                    Binary(*, Constant(3), Binary(+,
Constant(4), Constant(5))))
```

The second time through the loop, we consume `-` from the input and make a recursive call to `parse_exp`. This time, because the precedence of `-` is 45, the second argument to `parse_exp` will be 46.

Following our usual routine, we get the next factor (`3`) and the next token (`*`). Since the precedence of `*` exceeds the minimum precedence, we need to parse another sub-expression. We consume `*`, leaving `(4 + 5)`, then make yet another recursive call to `parse_exp`.

In this next call to `parse_exp`, we start by calling `parse_factor` as usual. This call will consume the rest of our input and return the AST node for `4 + 5`. To handle that parenthesized expression, `parse_factor` will need to recursively call `parse_exp` with the minimum precedence reset to zero, but we won't step through that here. At this point, there are no tokens left in our expression. Let's assume this is a valid C program and the next token is a semicolon. Since the next token isn't a binary operator, we just return the expression we got from `parse_factor`.

At the next level up, we construct the AST node for `3 * (4 + 5)` from the sub-expressions we've processed in this call. Once again, we peek at the next token, see that it isn't a binary operator, and return.

Finally, back in the original call to `parse_exp`, we construct the final expression from the left operand that we constructed in the first loop iteration (`1 * 2`), the current value of `next_token` (`-`), and the right operand that was just returned from the recursive call (`3 * (4 + 5)`). For the last time, we check the next token, see that it isn't a binary operator, and return.

Now that we've seen how to parse binary expressions with precedence climbing, you're ready to extend your own parser. Remember that you'll use precedence climbing to parse binary expressions, and recursive descent to parse all the other symbols in the grammar, including factors.

FURTHER READING ON PRECEDENCE CLIMBING

These blog posts helped me understand precedence climbing, and how it relates to similar algorithms that solve the same problem. You might find them helpful too.

- "Parsing expressions by precedence climbing," a blog post by Eli Bendersky, provides a good overview of the precedence climbing algorithm. It also covers right-associative operators, which I didn't discuss here. (*https://eli.thegreenplace.net/2012/08/02/parsing-expressions-by-precedence-climbing*)
- "Some problems of recursive descent parsers," also by Eli Bendersky, goes into more detail about how to handle binary expressions with a pure recursive descent parser. (*https://eli.thegreenplace.net/2009/03/14/some-problems-of-recursive-descent-parsers*)
- Andy Chu has written two useful blog posts on precedence climbing. The first, "Pratt Parsing and Precedence Climbing are the Same Algorithm" explores the fundamental similarities between these two approaches (*https://www.oilshell.org/blog/2016/11/01.html*). The second, "Precedence Climbing is Widely Used," discusses their differences (*https://www.oilshell.org/blog/2017/03/30.html*). These posts clarify some of the confusing terminology around different parsing algorithms.

Testing the Parser

The parser should be able to handle every valid test case in *tests/chapter_4/valid*, and raise an error on every invalid test case in *tests/chapter_4/invalid_parse*. To test your parser against the test cases from this chapter and the ones before it, run:

```
$ ./test_compiler /path/to/your_compiler --chapter 4 --stage
parse
```

Remember that the test suite only checks whether your compiler parses a program successfully or throws an error; it doesn't check that it produced the correct AST. In this chapter, it's especially easy to write a parser that appears to succeed but generates the wrong AST, so you might want to write your own tests to validate the output of your parser.

Extending TACKY Generation

Next, we need to update the stage that converts the AST to TACKY. We'll start by updating TACKY itself to include binary operations:

```
program = Program(function_definition)
function_definition = Function(identifier, instruction*
body)
instruction = Return(val)
            | Unary(unary_operator, val src, val dst)
            | Binary(binary_operator, val src1, val src2,
val dst)
val = Constant(int) | Var(identifier)
unary_operator = Complement | Negate
binary_operator = Add | Subtract | Multiply | Divide | Mod
```

Listing 4-8 Adding binary expressions to TACKY

The changes here are pretty straightforward: we've just added one new type of instruction to represent binary operations, and defined all the possible operators. Like unary operations, binary operations in TACKY can only be applied to constants and variables, not to nested sub-expressions. We can turn a binary expression into a sequence of TACKY instructions in almost exactly the same way that we handled unary expressions:

```
emit_tacky(e, instructions):
    match e with
    | --snip--
    | Binary(op, e1, e2) ->
        v1 = emit_tacky(e1, instructions)
        v2 = emit_tacky(e2, instructions)
        dst_name = make_temporary()
        dst = Var(dst_name)
        tacky_op = convert_binop(op)
        instructions.append(Binary(tacky_op, v1, v2, dst))
        return dst
```

Listing 4-9 Converting a binary operation to TACKY

We need to emit the TACKY instructions to calculate each operand, then emit the binary expression that uses those source values. The only difference from how we handled unary expression is that we're processing two operands instead of one.

NO, *YOU'RE* OUT OF ORDER!

In Listing 4-9, we generate code that evaluates the first operand, then the second operand, then the whole operation. Surprisingly, it would be just as correct to evaluate the second operand before the first. According to the C standard, sub-expressions of the same operation are *unsequenced*—they can be evaluated in any order. In the programs we can compile so far, it doesn't matter which operand we evaluate first; you'll get the same visible behavior either way. That's not the case in the following program:

```
#include <stdio.h>

int main() {
    return printf("Hello, ") + printf("World!");
}
```

You could compile this program with a C standard-compliant compiler, run it, and get either of the following outputs:

```
Hello, World!
World!Hello,
```

There are a few exceptions where the first operand must be evaluated first: the logical `&&` and `||` operators, which we'll cover next chapter; the conditional `?:` operator, which we'll cover a few chapters later; and the comma operator, which we won't implement.

If you're curious, the relevant part of the C18 standard is section 6.5, paragraphs 1-3. There's also a more readable explanation at *https://en.cppreference.com/w/c/language/eval_order.*

Testing TACKY Generation

The TACKY generator should be able to process every valid test case we've seen so far. You can test it with:

```
$ ./test_compiler /path/to/your_compiler --chapter 4 --stage
codegen
```

Extending Assembly Generation

The next step is converting TACKY into assembly. We'll need several new assembly instructions to handle addition, subtraction, multiplication, division, and the modulo operation. Let's talk through these new instructions.

Doing Arithmetic in Assembly

The instructions for addition, subtraction, and multiplication all take the form ***`op src, dst`***, where:

`op` is an instruction,

`src` is an immediate value, register, or memory address, and

`dst` is a register or memory address.

Each of these instructions applies ***`op`*** to ***`dst`*** and ***`src`***, storing the result in ***`dst`***. The instructions for addition, subtraction, and multiplication are add, sub, and imul, respectively. As usual, these instructions can take an l suffix if their operands are 32 bits, and a q suffix if their operands are 64 bits. Here's an example of each instruction:

Table 4-2 Assembly instructions for addition, multiplication, and division

Instruction	Meaning
`addl $2, %eax`	`eax = eax + 2`
`subl $2, %eax`	`eax = eax - 2`
`imull $2, %eax`	`eax = eax * 2`

Note that ***dst*** is the *first* operand in the corresponding mathematical expression; that means subl a, b computes b - a, not a - b.

These instructions are pretty easy to use and understand! In a perfect world, we could perform division in exactly the same way. But we don't live in a perfect world, which is why we're stuck with the idiv instruction.

Division is Weird

We'll use the idiv instruction to implement the division and modulo operations. Even though you need two numbers to perform division, it takes a single operand, which can't be an immediate. This operand is the divisor. (A reminder, in case you're like me and always get them mixed up: in a / b, a is the dividend and b is the divisor.) In its 32-bit form, idiv gets the other value it needs, the dividend, from the EDX and EAX registers, which it treats as a single, 64-bit value. It gets the most significant 32 bits from EDX and the least significant 32 bits from EAX. Unlike our other arithmetic instructions, division produces two results: the quotient and the remainder. The quotient is stored in the EAX register, and the remainder is stored in the EDX register. (The 64-bit version of idiv, which we'd write as idivq, uses RDX and RAX as the dividend instead of EDX and EAX.)

In order to use idiv, we need to turn a 32-bit dividend into a 64-bit value spanning both EDX and EAX. Whenever we need to convert a signed integer to a wider format, we'll use an operation called *sign extension*. This operation fills the upper 32 bits of the new, 64-bit value with the sign bit of the original 32-bit value.

Sign extending a positive number just pads the upper 32 bits with zeros. Sign-extending the binary representation of 3, for example, turns

```
00000000000000000000000000000011
```

into

```
0000000000000000000000000000000000000000000000000000000000000
0011
```

Both representations have the value 3; the second one just has more leading zeros. To sign extend a negative number, we fill the upper four bytes with ones, which converts -3 from

```
11111111111111111111111111111101
```

into

```
1111111111111111111111111111111111111111111111111111111111111
1101
```

Thanks to the magic of two's complement, the value of both of these binary numbers is -3. (If you're not clear on how this works, you can check out the further reading on two's complement from Chapter 3.)

The `cdq` instruction does exactly what we need here: it sign extends the value from EAX into EDX. If the number in EAX is positive, this instruction will set EDX to all zeros. If EAX is negative, this instruction will set EDX to all ones. Putting it all together, here's how you'd compute `9 / 2`, or `9 % 2`, in assembly:

```
movl    $2, %ebx
movl    $9, %eax
cdq
idiv    %ebx
```

The result of `9 / 2`, the quotient, will be stored in EAX. The result of `9 % 2`, the remainder, will be stored in EDX.

Now we've covered all the new instructions we'll need in this chapter: `add`, `sub`, `imul`, `idiv`, and `cdq`. Next, let's add these new instructions to the assembly AST and update the conversion from TACKY to assembly.

Converting TACKY to Assembly

Here's the updated assembly AST, with additions bolded:

```
program = Program(function_definition)
function_definition = Function(identifier name, instruction*
instructions)
instruction = Mov(operand src, operand dst)
            | Unary(unary_operator, operand)
            | Binary(binary_operator, operand, operand)
            | Idiv(operand)
            | Cdq
            | AllocateStack(int)
            | Ret
```

```
unary_operator = Neg | Not
binary_operator = Add | Sub | Mult
operand = Imm(int) | Reg(reg) | Pseudo(identifier) |
Stack(int)
reg = AX | DX | R10 | R11
```

Listing 4-10 Adding new instructions to the assembly AST

Since the addition, subtraction, and multiplication instructions all take the same form, we'll represent them all using the `Binary` instruction node. We'll also add instruction nodes for the new `idiv` and `cdq` instructions. We'll add the EDX register to the AST definition, since the `idiv` instruction uses it. We'll also add the R11 register to use along with R10 during the instruction fix-up pass.

Now we need to convert our new binary operations from TACKY to assembly. For addition, subtraction, and multiplication, we'll convert a single TACKY instruction into two assembly instructions:

```
Binary(op, src1, src2, dst)
```

becomes

```
Mov(src1, dst)
Binary(op, src2, dst)
```

Division is a little more complicated; we need to move the first operand into EAX, sign-extend it with `cdq`, issue the `idiv` instruction, and then move the result from EAX to the destination. So

```
Binary(Divide, src1, src2, dst)
```

becomes

```
Mov(src1, Reg(AX))
Cdq
Idiv(src2)
Mov(Reg(AX), dst)
```

The modulo operation looks exactly the same, except that we ultimately want to retrieve the remainder from EDX instead of retrieving the quotient from EAX. So

```
Binary(Mod, src1, src2, dst)
```

becomes

```
Mov(src1, Reg(AX))
Cdq
Idiv(src2)
Mov(Reg(DX), dst)
```

The `idiv` instruction can't operate on immediate values, so the assembly instructions for division and modulo won't be valid if `src2` is a constant. That's okay; we'll fix it during the instruction-rewriting pass. Table 4-3 summarizes the conversion from TACKY to assembly.

Table 4-3 Conversion from TACKY to Assembly

TACKY	**Assembly**
Top-level constructs	
`Program(function_definition)`	`Program(function_definition)`
`Function(name, instructions)`	`Function(name, instructions)`
Instructions	

`Return(val)`	`Mov(val, Reg(AX))`
	`Ret`

`Unary(unary_operator, src, dst)`	`Mov(src, dst)`
	`Unary(unary_operator, dst)`

`Binary(Divide, src1, src2, dst)`	**`Mov (src1, Reg(AX))`**
	`Cdq`
	`Idiv(src2)`
	`Mov(Reg(AX), dst)`

`Binary(Mod, src1, src2, dst)`	**`Mov (src1, Reg(AX))`**
	`Cdq`
	`Idiv(src2)`
	`Mov(Reg(DX), dst)`

`Binary(binary_operator, src1, src2, dst)`	**`Mov(src1, dst)`**
	`Binary(binary_operator, src2, dst)`

Operators	
`Complement`	`Not`
`Negate`	`Neg`
`Add`	**`Add`**
`Subtract`	**`Sub`**
`Multiply`	**`Mult`**
Operands	