Computer Systems VI: Compilers

Noah Singer, George Klees

Montgomery Blair High School

April 8, 2018

What are Compilers?

Definition (Compiler)

A program that translates code from a **source language** into a **target language**.

- Usually we're dealing with from some **high-level language** (e.g. C, C++) to **assembly language**
- Assembly is then assembled by an assembler

Hello, Compiler World

Here's a simple C program.

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("Hello, compiler world!\n");
    return 0;
}
```

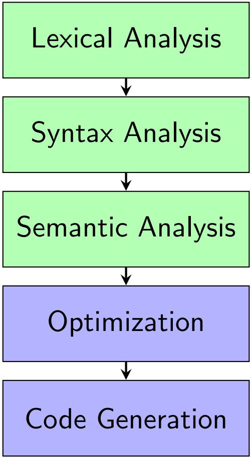
Next, we're going to see what kind of assembly this compiles to.

Hello, Assembly World

```
main:
  sh %rbp
  mov %rsp,%rbp
  sub $0x10,%rsp
  mov %edi,-0x4(%rbp)
  mov %rsi,-0x10(%rbp)
  mov $0x40060c,%edi
  callq 4003f0 <puts@plt>
  mov $0x0,%eax
  leaveq
  retq
```

This begs the question: **What does the compiler do in order to transform code from C to assembly?**

Stages of a Compiler



The five traditional stages of a compiler.

Stages of a Compiler

1. **Lexical analysis.** The source code is split into **tokens** like integers (INT), identifiers (IDEN), or keywords (e.g. IF).
2. **Syntax analysis.** The source code is analyzed for structure and parsed into an **abstract syntax tree** (AST).
3. **Semantic analysis.** Various intermediate-level checks and analyses like **type checking** and making sure variables are declared before use.
4. **Optimization.** At this point, the code is usually converted into some platform independent **intermediate representation** (IR). The code is optimized first platform-independently and then platform-dependently.
5. **Code generation.** Target language code is generated from the IR.

Lexical Analysis

- Programs which perform lexical analysis are called **lexers** for short
- Two main stages:
  1. The **scanner** splits the input into pieces called **lexemes**
  2. The **evaluator** creates tokens from lexemes, in some cases assigning tokens a **value** based on their lexeme
- Whitespace and comments are ignored
- Lexical analysis is considered “solved” since efficient algorithms have been discovered
  - Programs called a **lexer generators** exist which will automatically create lexers
  - Most common lexer generator is called `flex` (new version of `lex`)

Guessing Game

```
printf("Guess a number between 1 and 100!");
int num = 21;

// Read an initial guess and then keep guessing
int guess;
scanf("%d\n", &guess);

while (guess != num)
{
    printf("Guess again!");
    scanf("%d\n", &guess);
}

// They got it right
printf("Good job! You got it!");
```

Guessing Game: Tokens

IDEN<printf> LPAREN STRLIT<Guess a number between 1 and 100!> RPAREN SEMI IDEN<int> IDEN<num> ASSIGN INT<21> SEMI IDEN<int> IDEN<guess> SEMI IDEN<scanf> LPAREN STRLIT<%d\n> COMMA UNARY<&> IDEN<guess> RPAREN SEMI WHILE LPAREN IDEN<guess> BINARY<!=> RPAREN LCURLY IDEN<printf> LPAREN STRLIT<Guess again!> RPAREN SEMI IDEN<scanf> LPAREN STRLIT<%d\n> COMMA UNARY<&> IDEN<guess> ENDWHILE SEMI RPAREN IDEN<printf> LPAREN STRLIT<Good job! You got it!> RPAREN SEMI

Primer on Regular Expressions

Let's take a look at **regular expressions** or **regex**, which are a useful tool for creating lexers. Regular expressions allow programmers and mathematicians to express "patterns" that encompass certain groups of strings.

- + is a unary postfix operator denoting "one or more"
- ? is a unary postfix operator denoting "zero or one"
- $*$ is a unary postfix operator denoting "zero or more"
- (and) can be used to group things together for precedence, just like in normal arithmetic
- [and] can be used for "character classes"(e.g. [0-9])
- | is an infix binary operator denoting "or"

Regular Expression Practice

1. A letter of the alphabet (uppercase or lowercase)

Regular Expression Practice

1. A letter of the alphabet (uppercase or lowercase)
2. `bat` or `cat`

Regular Expression Practice

1. A letter of the alphabet (uppercase or lowercase)
2. `bat` or `cat`
3. `ab`, `abab`, `ababab`, etc.

Regular Expression Practice

1. A letter of the alphabet (uppercase or lowercase)
2. `bat` or `cat`
3. `ab`, `abab`, `ababab`, etc.
4. `15`, `3.70`, `-10.801`, `-5.2E7`, etc. `6.9E-2`

Regular Expression Practice

1. A letter of the alphabet (uppercase or lowercase)
2. `bat` or `cat`
3. `ab`, `abab`, `ababab`, etc.
4. `15`, `3.70`, `-10.801`, `-5.2E7`, etc. `6.9E-2`
5. Regex to match valid C/Java variable names

Regular Expression Practice

1. A letter of the alphabet (uppercase or lowercase)
2. `bat` or `cat`
3. `ab`, `abab`, `ababab`, etc.
4. `15`, `3.70`, `-10.801`, `-5.2E7`, etc. `6.9E-2`
5. Regex to match valid C/Java variable names
6. Regex to match only binary strings divisible by three

Regular Expression Practice

1. A letter of the alphabet (uppercase or lowercase)
2. `bat` or `cat`
3. `ab`, `abab`, `ababab`, etc.
4. `15`, `3.70`, `-10.801`, `-5.2E7`, etc. `6.9E-2`
5. Regex to match valid C/Java variable names
6. Regex to match only binary strings divisible by three
7. `()`, `(())`, `((()))`, `(((())))`, etc.

Regular Expressions for Lexical Analysis

- The lexer is based off of a **lexical grammar** that contains a pattern for each type of token
- Efficient parsing can then be completed using **deterministic finite automata** or **nondeterministic finite automata**
- The evaluator assigns a value to some tokens (e.g. INT tokens) based on their corresponding lexeme

Sample lexical grammar:

IDEN: `[a-zA-Z_][a-zA-Z0-9_]*`
INT: `(\+|-)?[0-9]+`
WHILE: `while`

and more!

Syntax Analysis

- As we've seen, regular expressions and lexical analysis by themselves aren't capable of encompassing the full complexity of programming languages
- For this, we need syntax analysis, also known as **parsing**
- Programs that create parsers are known as **parser generators**, the most popular of which is `bison` (new form of `yacc`); `bison` and `flex` play together very nicely
- Generally parses the stream of tokens into an abstract syntax tree or **parse tree**, difference being that abstract syntax tree doesn't include every detail of source code syntax
- Usually accomplished with a **context-free grammar** (CFG)
- Parsing can be either **bottom-up** or **top-down**
- To understand this, let's take a look at some formal language theory!

Formal Language Theory

- A **language** is a set (finite or infinite) of **words** over some **alphabet** consisting of **letters** Σ
- Each language has **syntax**, which describes how it looks, and **semantics**, which describes what it means
- A language is often defined by some set of rules and constraints called a **grammar**
  - Consists mostly of **productions**, which are rules mapping some **symbols** to the union of one or more strings of symbols
- Example language (this only only has one production): $E \to (E) \mid E * E \mid E/E \mid E + E \mid E - E \mid \text{INT}$
  - What is this?
- In context-free languages, a symbol can always be replaced using a production, regardless of its context (left hand side is only one symbol)

Guessing Game: Revisited

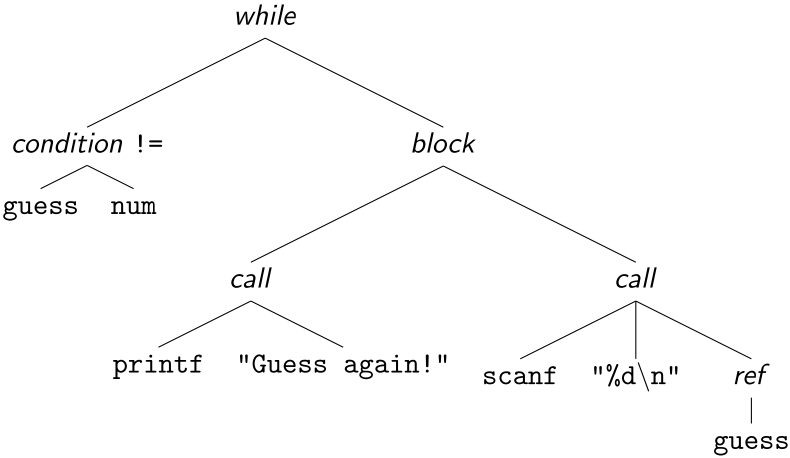

```c
printf("Guess a number between 1 and 100!");
int num = 21;

// Read an initial guess and then keep guessing
int guess;
scanf("%d\n", &guess);

while (guess != num)
{
    printf("Guess again!");
    scanf("%d\n", &guess);
}

// They got it right
printf("Good job! You got it!");
```

Guessing Game: Abstract Synax Tree



Guessing Game: Model Grammar

A vastly oversimplified model grammar for C that somewhat works with our example.

$$
\begin{aligned}
&\mathit{while} \rightarrow \mathit{condition}\ \mathit{block}\\
&\mathit{condition} \rightarrow \mathit{expr}\ \text{CMP}\ \mathit{expr}\\
&\mathit{expr} \rightarrow (\mathit{expr}) \mid \text{UNARY}\ \mathit{expr} \mid \mathit{callexpr}\ \text{BINARY}\ \mathit{expr} \mid \text{IDEN} \mid \text{STRLIT}\\
&\mathit{block} \rightarrow \mathit{statement}\ \mathit{block}\\
&\mathit{statement} \rightarrow \mathit{call}\ \text{SEMI} \mid \mathit{assign}\ \text{SEMI}\\
&\mathit{call} \rightarrow \text{IDEN LPAREN}\ \mathit{args}\ \text{RPAREN}\\
&\mathit{assign} \rightarrow \text{IDEN IDEN ASSIGN}\ \mathit{expr}\\
&\mathit{args} \rightarrow \mathit{expr} \mid \mathit{expr}\ \text{COMMA}\ \mathit{args}
\end{aligned}
$$

Notes about Formal Languages

- Various methods exist to recognize and parse various kinds of formal languages
  - **Recursive-descent** parsing is popular because it's conceptually simple but has fallen out of favor because it's slow and inefficient
  - **LALR-1** (lookahead-1 left-right) parsing is used in most modern programming languages, but it's relatively complex
- Some grammars may be **ambiguous**, meaning that there are multiple ways to produce the same string (we must specify things like order of operations and associativity)
- There are various related classes of languages (context-free is one of them)
- Formal language theory is intimately related to natural language processing, linguistics, automata theory, and theory of computation
- Anyone sensing another lecture?

Semantic Analysis

- Once we've figured out the syntactic structure of our program, we still have more work to do before actually generating code
- Semantic analysis basically includes doing a bunch of things to work out the "meaning" of our code before we actually generate output, including:
  - **Type checking**: assigning every expression a type and ensuring that all types are correct and there are no type mismatches (this is done very differently in different languages)
  - Checking for multiple definitions of functions and variables
  - Checking that functions and variables can each be matched to a definition
- In general, semantic analysis adds information to the abstract syntax tree, creating an **attributed abstract syntax tree**

Type Checking

- There are multiple ways that type checking can work in programming languages
- **Static type checking** is type checking done at compile time
- **Dynamic type checking** is type checking done at runtime
- Static type checking is faster and safer (makes better guarantees), but it's less flexible and doesn't allow some useful features
- There are three major type systems
  - **Structural typing**, in which objects' types are defined by their actual structure and not their name
  - **Nominative typing**, in which objects' types are defined by explicit declaration
  - **Duck typing**, in which objects' types aren't checked but rather if they possess some functionality requisite in some scenario: "When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."

Optimization: definitions

Definition (Optimizing compiler)

A compiler that is built to minimize or maximize some attributes of a program's execution, generally through a sequence of **optimizing transformations**.

- Usually we're minimizing time, but also we sometimes want to minimize memory, program size, or power usage, especially in mobile devices
- Many optimization problems are NP-HARD or even undecidable, so in general, optimizing compilers use many heuristics and approximations and often don't come up with a near to ideal program
- Optimizations may be either **platform-dependent** or **platform-independent**

Types of optimization

Definition (Basic Block)

A code sequences that contains no jumps or branches besides the entrance into the sequence and exit from the sequence.

1. **Peephole.** Looks at a few instructions at a time, typically micro-optimizing small instruction sequences.
2. **Local.** Contained within one basic block.
3. **Loop.** Acts on a loop.
4. **Global.** Between multiple basic blocks in a single function.
5. **Interprocedural/whole-program.** Between multiple functions in a program.

Common techniques

1. **Strength reduction.** Complex computations are reduced to less "expensive", but equivalent, computations in order to save computation time (or power consumption or whatever is being optimized).
2. **Avoid redundancy and eliminate dead stores/code.** Eliminate code that isn't used, variables that aren't used, and calculating the same thing multiple times.
3. **Elimination of jumps.** Jumps, loops, and function calls slow down programs significantly, so in some cases, for example, loops are unrolled at the cost of increasing binary program size.
4. **Fast path.** When there is a branch with two choices in a program, and one of them is much more common than the other, that one can automatically be assumed, and then "undone" if the condition turns out to be false.

Data-flow optimization

1. **Induction variable analysis**

```
for (int i = 0; i < 10; i++)
{
        printf("%d\n", 8*i+2);
}
```

Data-flow optimization

1. **Induction variable analysis**

```
for (int i = 0; i < 10; i++)
{
        printf("%d\n", 8*i+2);
}
```

2. **Constant propagation and constant folding**

```
int a = 8 * 3 + 2 / 7;
printf("%d\n", a+5);
```

Data-flow optimization

1. **Induction variable analysis**

```
for (int i = 0; i < 10; i++)
{
        printf("%d\n", 8*i+2);
}
```

2. **Constant propagation and constant folding**

```
int a = 8 * 3 + 2 / 7;
printf("%d\n", a+5);
```

3. **Common subexpression elimination**

```
int a = (c * 3) + 47;
int b = (c * 3) % 2;
```

Data-flow optimization

1. **Induction variable analysis**

```
for (int i = 0; i < 10; i++)
{
        printf("%d\n", 8*i+2);
}
```

2. **Constant propagation and constant folding**

```
int a = 8 * 3 + 2 / 7;
printf("%d\n", a+5);
```

3. **Common subexpression elimination**

```
int a = (c * 3) + 47;
int b = (c * 3) % 2;
```

4. **Dead store elimination**

Other optimizations

1. **Tail call elimination**

```
int fib(int product, int count, int max)
{
        if (count > max) return product;
        else return fib(product * count,
            count + 1, max);
}
```

2. **Strength reduction of multiplication and division**
3. **Function inlining**

Intermediate representation

Definition (Intermediate language)

A language used to describe code running on a theoretical, simple machine

- The AST is translated into an intermediate language, which the machine code is generated from
- Most intermediate languages (such as the common three-address code) have unlimited variables and can only do a single operation in one line

Intermediate representation

```
for (int i = 0; i < 8; i++)
        printf("%d\n", 2 * a / b - 7 * c);
```

```
r0 := 0
in_loop:
  r1 := 2 * a
  r2 := r1 / b
  r3 := 7 * c
  r4 := r3 - r2
  r5 := "%d\n"
  printf(r1, r5)
  r0 := r0 + 1
  jmp_nlt r0 8 done
done:
  exit
```

Registers, caches, and RAM

Definition (Memory)

A piece of hardware that stores binary data.

1. **Registers.** Internal CPU memory
   - Each register is a fixed size, and assigned a number
2. **Cache.** Multi-level buffer between CPU and RAM
3. **RAM.** External memory
   - Every byte (8-bits) assigned a **memory address**

Cache architecture

- Multiple cache levels (L1, L2, L3), with lower numbers having less memory, being faster, and shared among fewer CPUs
- A single cache consists of several **cache blocks**, holding data from RAM
- When the CPU accesses RAM, it first tries each level of the cache in ascending order (L1, L2, ...) before going to RAM
- Data not being found at a certain level is a **cache miss**, meaning the data must be retrieved from a higher level (performance penalty) and moved to the lower levels
- When a new block is read after a cache miss, a less-recently used block may be **evicted**

Memory access latency (2016)

1. Single clock cycle: <1 ns
2. L1 cache reference: 1 ns
3. L2 cache reference: 4 ns
4. Main memory reference: 100 ns
5. Read 1MB from disk: 1ms
6. Send packet to the Netherlands over network: 150 ms

Adapted from Colin Scott at UC Berkeley.
http://www.eecs.berkeley.edu/~rcs/research/interactive_latency.html

Locality of reference

Definition (Locality of reference)

Locality of reference occurs when memory accesses are correlated and close together.

1. **Temporal locality:** If a memory location is accessed, it will probably be accessed in the near-future
2. **Spatial locality:** If a memory location is accessed, nearby locations will probably be accessed in the near-future

- The memory hierarchy (registers, cache, RAM) is based on locality commonly used variables are kept in faster memory

Optimizing register use

- Registers are faster than cache or RAM, so all local variables should be in registers if possible
- If there are more variables than registers, the compiler must allocate the variables to either registers or the stack
  1. Construct a graph; variables are nodes, **interference edges** connect simultaneously-used variables, and **preference edges** connect one variable that's set to another
  2. With K registers available, **assign each node a color** such that: no nodes sharing an interference edge are the same color, and nodes sharing preference edges are the same color if possible
  3. Color of each variable is its register assignment

Optimizing the cache

- The idea of a cache is based on locality of reference, so code should take advantage of it
- Ensure spatial locality (and minimize cache misses) by keeping related data together
  - Keep code and data compact
  - Put data that will be accessed at similar times in the same cache block

The importance of spatial locality

- How would you multiply matrix A (m x n) by matrix B (p x q)?

```
for (int rA = 0; rA < n; rA++)
  for (int cB = 0; cB < p; cB++)
    for (int rB = 0; rB < q; rB++)
      C[rA][cB] += A[rA][rB] * B[rB][cB];
```

```
for (int rA = 0; rA < n; rA++)
  for (int rB = 0; rB < q; rB++)
    for (int cB = 0; cB < p; cB++)
      C[rA][cB] += A[rA][rB] * B[rB][cB];
```

Loop optimization

1. **Loop fission/distribution.** Split a loop into multiple sequential loops to improve locality of reference.
2. **Loop fusion/combination.** Combine multiple sequential loops (with the same iteration conditions) to reduce overhead.
3. **Loop interchange.** Switching an inner and outer loop in order to improve locality of reference.
4. **Loop-invariant code motion.** Move code that calculates some value that doesn't change to outside the loop.
5. **Loop unrolling.** Replace a loop that iterates some fixed N times with N copies of the loop body.