Language Translators

Ronghui Gu
Spring 2019
Columbia University

* Course website: https://www.cs.columbia.edu/~rgu/courses/4115/spring2019
** These slides are borrowed from Prof. Edwards.
What is a Translator?

A programming language is a notation that a person and a computer can both understand.

- It allows you to express what is the **task** to compute
- It allows a computer to **execute** the computation task

A translator translates what you express to what a computer can execute.
• **Pros**: translation is done once and for all; optimize code and map identifiers at compile time.

• **Cons**: long compilation time; hard to port.
• **Pros**: source code distribution; short development cycle.

• **Cons**: translation is needed every time a statement is executed; lack optimization; map identifiers repeatedly.
• **Pros**: bytecode is highly compressed and optimized; bytecode distribution.

• **Cons**: compilation overhead + interpreter overhead.
Just-In-Time Compiler

- **Pros**: compile and optimize many sections just before the execution; bytecode distribution.
- **Cons**: compilation overhead + warm-up overhead.
Language Speeds Compared

Native code compilers
Just-in-time compilers
Bytecode interpreters

ATS
C++ GNU g++
C GNU gcc
Java 6 steady state
Ada 2005 GNAT
Haskell GHC
Scala
Java 6 -server
Lua LuaJIT
Fortran Intel
OCaml
F# Mono
C# Mono
Go 6g 8g
Racket
Lisp SBCL
JavaScript V8
Erlang HiPE
Lua
Smalltalk VisualWorks
Java 6 -Xint
Python CPython
Python 3
Ruby 1.9
Mozart/Oz
PHP
Perl

Source: http://shootout.alioth.debian.org/
Compilation Phases
```c
int avg(int a, int b) {
    return (a + b) / 2;
}
```
Compilation Phases

int avg (int a, int b) ...

Lexical Analysis

Syntax Analysis

Semantic Analysis

Intermediate Code Generation

Optimization

Code Generation

0101110101...

front-end

middle-end

back-end
What the Compiler Sees

```c
int avg(int a, int b)
{
    return (a + b) / 2;
}
```

```c
int SP avg(intSP a, intSP b)
{
    return (a + b) / 2;
}
```

Just a sequence of characters
Lexical Analysis Gives Tokens

```c
int avg(int a, int b)
{
    return (a + b) / 2;
}
```

- A stream of tokens; whitespace, comments removed.
- Throw errors when failing to create tokens: malformed strings or numbers or invalid characters (such as non-ASCII characters in C).
Syntax Analysis

```
int avg (int a, int b) ...
```

Lexical Analysis

Syntax Analysis

Semantic Analysis

Intermediate Code Generation

Optimization

Code Generation

/zero.osf/one.osf/zero.osf/one.osf/one.osf/one.osf/zero.osf/one.osf/zero.osf/one.osf...

front-end

middle-end

back-end
Syntax Analysis Gives an Abstract Syntax Tree

```
int avg(int a, int b)
{
    return (a + b) / 2;
}
```

- Syntax analysis will throw errors if "}" is missing. Lexical analysis will not.
Semantic Analysis

`int avg (int a, int b) ...

Lexical Analysis

Syntax Analysis

Semantic Analysis

Intermediate Code Generation

Optimization

Code Generation

0101110101...
Semantic Analysis: Resolve Symbols; Verify Types

```
func avg(args arg a, arg b)
    return a + b + 2
```

Symbol Table

- `int a`
- `int b`
Intermediate Code Generation

- **Lexical Analysis**
- **Syntax Analysis**
- **Semantic Analysis**
- **Intermediate Code Generation**
- **Optimization**
- **Code Generation**

```
int avg (int a, int b) ...
```

The process is divided into three stages:
- **Front-end**
- **Middle-end**
- **Back-end**
int avg(int a, int b)
{
    return (a + b) / 2;
}

Idealized assembly language w/ infinite registers

avg:
    t0 := a + b
    t1 := 2
    t2 := t0 / t1
    ret t2
int avg (int a, int b) ...

Lexical Analysis

Syntax Analysis

Semantic Analysis

Intermediate Code Generation

Optimization

Code Generation

0101110101...
Optimization

avg :
  t0 := a + b
  t1 := 2
  t2 := t0 / t1
  ret t2

avg :
  t0 := a + b
  t2 := t0 / 2
  ret t2
int avg (int a, int b) ...

Lexical Analysis

Syntax Analysis

Semantic Analysis

Intermediate Code Generation

Optimization

Code Generation

0101110101...
avg:
  t0 := a + b
  t2 := t0 / 2
  ret t2

avg:
  pushl %ebp # save BP
  movl %esp,%ebp
  movl 8(%ebp),%eax # load a from stack
  movl 12(%ebp),%edx # load b from stack
  addl %edx,%eax # a += b
  shr $1,%eax # a /= 2
  ret