1. Introduction

NYU Courant Institute
Compiler Construction (CSCI-GA.2130-001)

Acknowledgments

Adapted from CSCI-GA.2130-001 slides
by Eva Rose and Kristoffer Rose.

1. Overview
2. Language Processors
3. Structure of a Compiler
4. Structure of the Course

Compiler

source program $\rightarrow$ COMPILER $\rightarrow$ target program

- semantically equivalent programs,
- source programs: typically high level,
- target programs: typically assembler or object/machine code,

object code $\neq$ object-oriented

Compiler

source program $\rightarrow$ COMPILER $\rightarrow$ target program

- semantically equivalent programs,
- source programs: typically high level,
- target programs: typically assembler or object/machine code,

object code $\neq$ object-oriented

Compiler

source program $\rightarrow$ COMPILER $\rightarrow$ target program

- semantically equivalent programs,
- source programs: typically high level,
- target programs: typically assembler or object/machine code,

object code $\neq$ object-oriented

Roles of a Compiler

- allow programming at an understandable abstraction level but execution based on low-level code;
- allow programs to be written in machine-independent languages but execution based on machine-specific code;
- help in verifying software
- early discovery of programming errors
- provide automatic code optimization.

Some Historical Highlights

- Grace Hopper coins the concept and writes the first compiler in 1952.
- John W. Backus presents the first formally based compiler (FORTRAN) in 1957.
- Frances E. Allen and John Cocke introduce most of the abstract concepts in compiler optimization and parallel compilers we use today; early 1960s.
- Alfred V. Aho and Jeffrey D. Ullman (and others) formalize parsing in 1960s and 70s (finite automata, context-free grammars).
- Donald Knuth publishes attribute grammars in 1968, which defines modern compiler construction methodology.

Meta-language: a language to talk about another language

Language Processors

Compiler: a program (written in a meta-language) that translates a program into a semantically equivalent program.

Interpreter: a program (written in a meta-language) for executing another program.

- Usually compilers faster than interpreters.
- Interpreters usually better at error diagnostics.

Language Processors

Compiler: a program (written in a meta-language) that translates a program into a semantically equivalent program.

Interpreter: a program (written in a meta-language) for executing another program.

- Usually compilers faster than interpreters.
- Interpreters usually better at error diagnostics.

Language Processors

Compiler: a program (written in a meta-language) that translates a program into a semantically equivalent program.

Interpreter: a program (written in a meta-language) for executing another program.

- Usually compilers faster than interpreters.
- Interpreters usually better at error diagnostics.

Interpreters

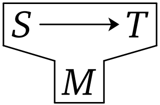
Interpreter diagrams, I-diagrams:

$$\boxed{\begin{array}{c} S \\ M \end{array}}$$

- S – *Source* language
- M – *Meta* or *Implementation* language

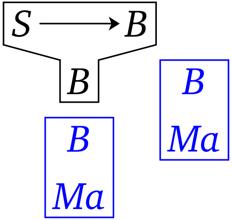
Compilers

Compiler diagrams, T-diagrams:



- *S* – compiled *Source* language
- *T* – generated *Target* language
- *M* – *Meta* or *Implementation* language

Hybrid: The Java Compiler



- S – compiled *Source* language
- B – intermediate *Bytecode* language
- Ma – actual *Machine* language

Some common examples

- Compiled languages: C, C++, Haskell, ML, (Java, C#) ...
- Interpreted languages: Python, Javascript, (Java) ...

Language-Processing System

Preprocessor: expands “macros” and combines source program modules.

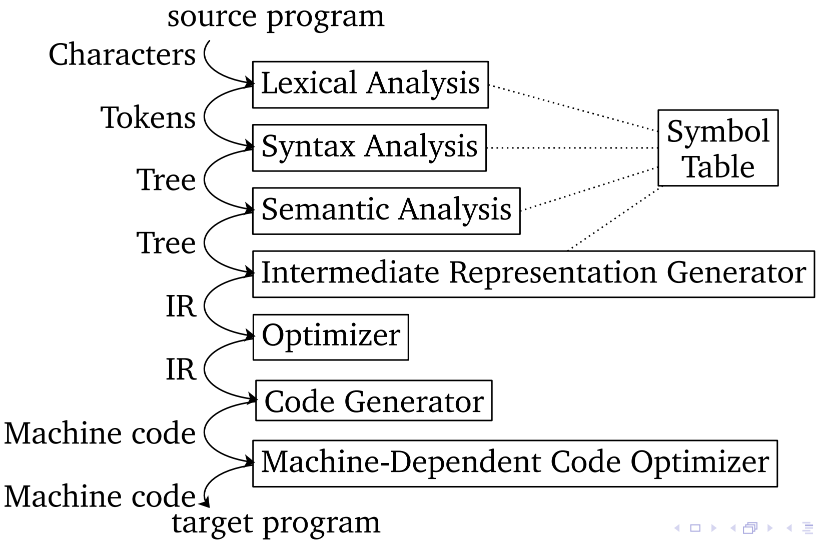
Compiler: translates source language to symbolic (assembler) machine code.

Assembler: translates symbolic machine code to relocatable binary code.

Linker: resolves links to library files and other relocatable object files.

Loader: combines executable object files in memory.

Transformation phases



Example

Source program arrives as stream of characters:

```
position = initial + rate * 60
```

Lexemes

Lexeme is the smallest meaningful entity of a language.

The lexemes here: `position`, `=`, `initial`, `+`, `rate`, `*`, and `60`.

Lexical Analysis

```
position = initial + rate * 60
```

scanned into list of tokens, one for each lexeme:

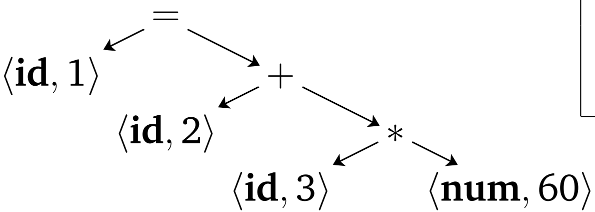
$\langle \mathbf{id}, 1\rangle \ \langle = \rangle \ \langle \mathbf{id}, 2\rangle \ \langle + \rangle \ \langle \mathbf{id}, 3\rangle \ \langle * \rangle \ \langle \mathbf{num}, 60\rangle$

1	`position`
2	`initial`
3	`rate`

Syntax Analysis

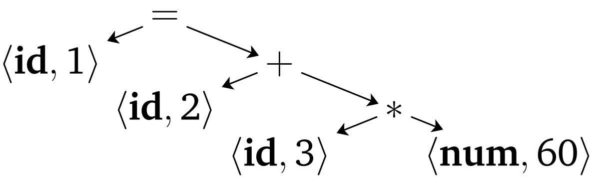
$$\langle \mathbf{id}, 1\rangle \quad \langle = \rangle \quad \langle \mathbf{id}, 2\rangle \quad \langle + \rangle \quad \langle \mathbf{id}, 3\rangle \quad \langle * \rangle \quad \langle \mathbf{num}, 60\rangle$$

parsed into syntax tree (using *precedence*):



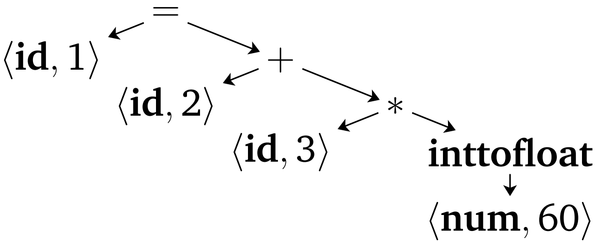
1	`position`
2	`initial`
3	`rate`

Semantic Analysis

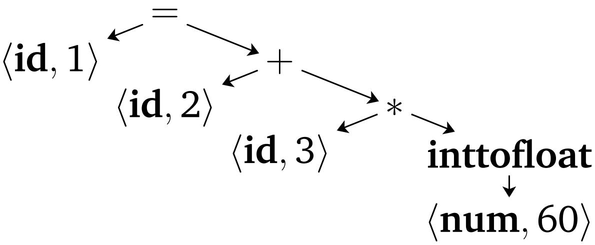


1	`position`
2	`initial`
3	`rate`

enriched with semantic information (explicit type conversion):



Intermediate Representation Generation



1	position
2	initial
3	rate

typically translated to intermediate code:

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Optimization

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

1	position
2	initial
3	rate

optimized to:

```
t1 = id3 * 60.0
id1 = id2 + t1
```

Good target code: faster, shorter, using less power.

Code Generation

1	position
2	initial
3	rate

```
t1 = id3 * 60.0
id1 = id2 + t1
```

generates (performing same task):

```
LDF  R2,  id3
MULF R2, R2, #60.0
LDF  R1, id2
ADDF R1, R1, R2
STF  id1, R1
```

Symbol Table

Records are <variable name, attributes>.

Attributes:

- storage information,
- type,
- scope (static/lexical, dynamic),
- procedure names: number and types of arguments,
- argument passing (by value or by reference)
- return type.

Design principle: find, store, retrieve records quickly.

Compiler-Construction Tools

Commonly used tools:

- scanner generators: token description $\rightarrow$ lexical analyser,
- parser generators: grammar $\rightarrow$ syntax analyser,
- syntax-directed translation engines: syntax tree $\rightarrow$ IR,
- code-generator generators: translation rules $\rightarrow$ code generator,
- data-flow engines: data-flow information analyzers.

Compiler generators: integrated set of the above.

1. Overview
2. Language Processors
3. Structure of a Compiler
4. **Structure of the Course**

Course Description

Standard compilers course following the Dragon Book.

- 11 lectures and homework assignments.
- 1 special topic.
- 2 exams.
- Semester-long programming project.

Grading

- 15% homework.
- 15% midterm exam.
- 25% final exam.
- 45% project.

Project

Implement fully functional compiler:

- Source language: ChocoPy.
- Target language: RISC-V assembly.
- Implementation language: Java.
- Logistics: Team effort; three parts; code and write-up.

ChocoPy

A dialect of Python designed at UC Berkeley for teaching compilers: `https://chocopy.org/`.

- Familiar: Can be executed by Python.
- Statically typed: Enforces Python type annotations.
- Expressive: Supports lists, classes, and nested functions.

ChocoPy Example

```
def contains(items:[int], x:int) -> bool:
    i:int = 0
    while i < len(items):
        if items[i] == x:
            return True
        i = i + 1
    return False
```

RISC-V

- Reduced instruction set computers (RISC) use a small set of general instructions.
- RISC-V is an open-source architecture based on RISC.
- Has offline and online simulators.

RISC-V Example

```
.globl $contains
$contains:
addi sp, sp, -@contains.size    # Space for stack frame
sw ra, @contains.size-4(sp)     # Return address
sw fp, @contains.size-8(sp)     # Control link
addi fp, sp, @contains.size     # New fp is at old SP
li a0, 0                        # Load integer literal 0
sw a0, -12(fp)                  # Local variable i
j label_6                       # Jump to loop test
```

Implementation Language

- Java: $\sim$ 5 KLOC given; another $\sim$ 5 KLOC to write.
- Will use lexer and parser generators (JFlex and CUP).
- Only use another language if you seek challenge.

Logistics

- Working in 3-4-person teams.
- Three milestones: parser; type checker; code generator.
- Submit code and write-up.

Project Review

Implement fully functional compiler:

- Source language: ChocoPy.
- Target language: RISC-V assembly.
- Implementation language: Java.
- Logistics: Team effort; three parts; code and write-up.

Project Challenges

- Volume and complexity of work.
- Need for independent investigation.
- Software-engineering challenges.
- Team and project management.

Thank you!