Compilers Construction

BS(CS) 2019-2023

Monday/Tuesday 12:30–14:00

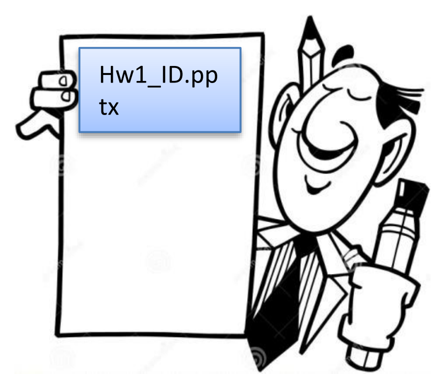
Instructor: Zeeshan Abbas
BS Computer Science (KIU Gilgit)
MS Computer Science (SEU China)
zeeshanabbas5@hotmail.com

Tentative Lecture Grading Policy

Class attendance	5%
Homework + Project/Presentation	15%
Final exam	50%
Midterm exam	30%

- Lecture representative is responsible for collection.

- The **FORMAT** of your PPT
file name **hw1_ID.ppt** or **hw1_ID.pdf**

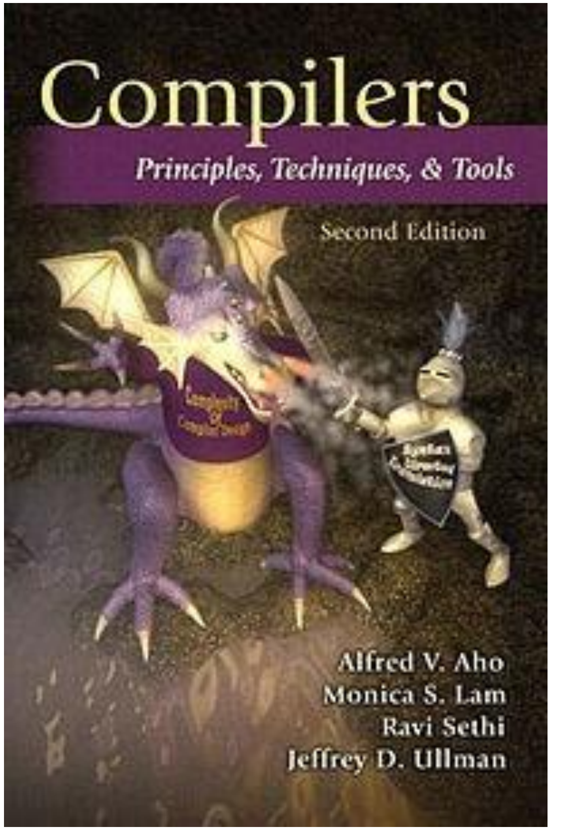


Grouping

- ➢10 Small groups
- ➢Each small group: 2-3 students
- ➢Project Presentation: Present by a small group at the end of the semester.
- ➢Homework: Individual students (2 students per lecture) present their homework.

Text books

- The Purple Dragon Book
- Aho, Lam, Sethi & Ullman
- Not required
  - But a useful reference

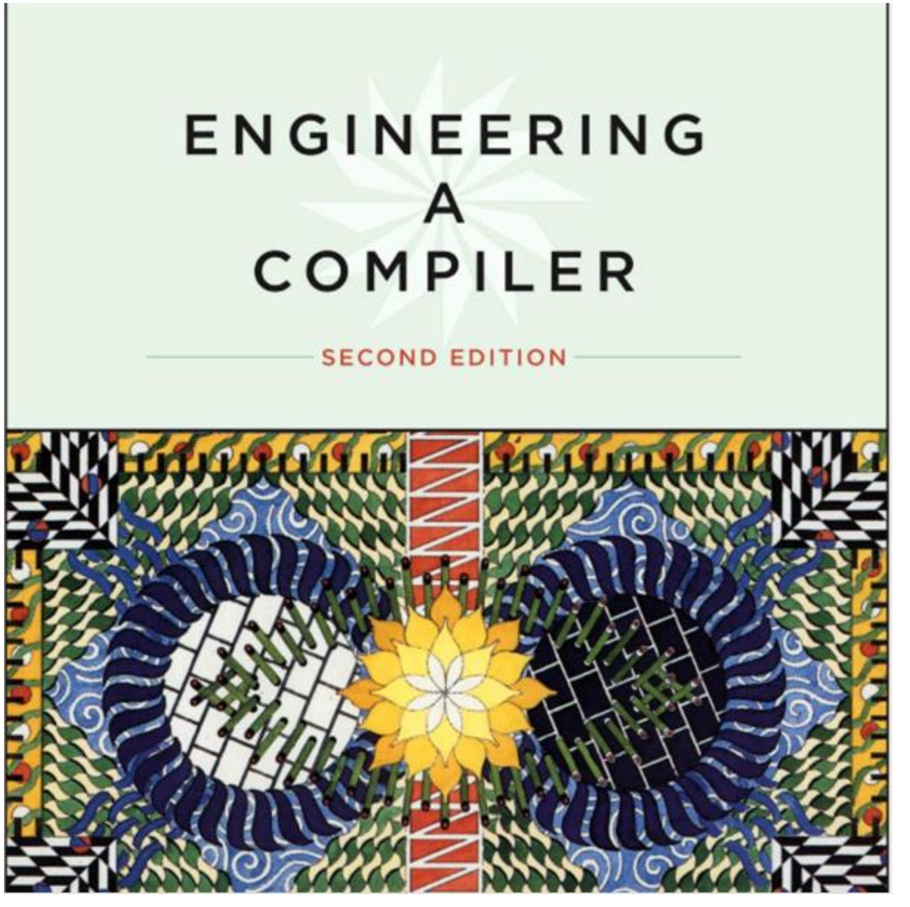


Text books

- Engineering a Compiler

- Keith D. Cooper, Linda Torczon

- Not required
  - But a useful reference



Course Goal

- Open the lid of compilers and see inside
  – Understand what they do
  – Understand how they work
  – Understand how to build them

- Correctness over performance
  – Correctness is essential in compilers
  – They must produce correct code
  – CS143 is more like CS103+CS110 than CS107
  – Other classes focus on performance (CS149, CS243)

History of High-Level Languages

- 1954: IBM develops the 704

- Problem
  - Software costs exceeded hardware costs!

- All programming done in assembly

The Solution

- Enter “Speedcoding”
- An interpreter
- 300 bytes = 30% of total memory
- Ran 10-20 times slower than hand-written assembly

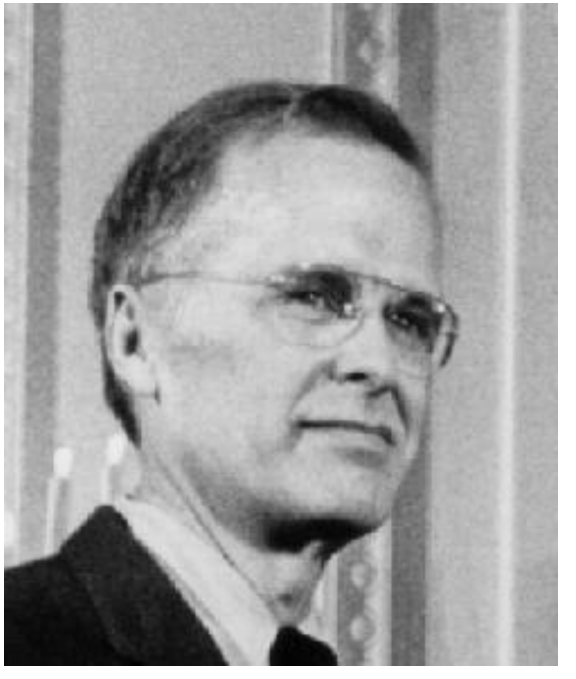
FORTRAN I

- Enter John Backus

- Idea
  - Translate high-level code to assembly

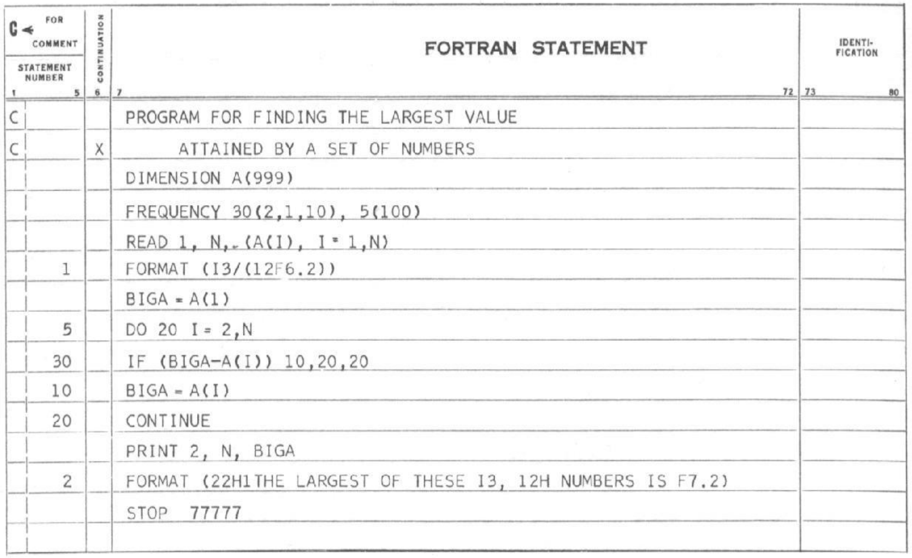
  - Many thought this impossible

  - Had already failed in other projects



FORTRAN I (Cont.)

- 1954-7
  - FORTRAN I project
- 1958
  - >50% of all software is in FORTRAN
- Development time halved
- Performance close to hand-written assembly!



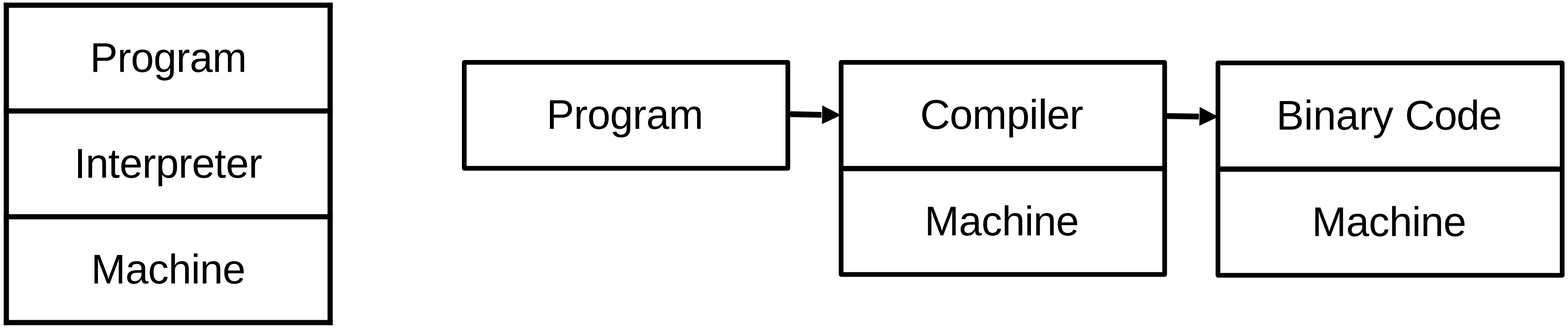

```
C     PROGRAM FOR FINDING THE LARGEST VALUE
C    X     ATTAINED BY A SET OF NUMBERS
      DIMENSION A(999)
      FREQUENCY 30(2,1,10), 5(100)
      READ 1, N, (A(I), I = 1,N)
    1 FORMAT (I3/(12F6.2))
      BIGA = A(1)
    5 DO 20 I = 2,N
   30 IF (BIGA-A(I)) 10,20,20
   10 BIGA = A(I)
   20 CONTINUE
      PRINT 2, N, BIGA
    2 FORMAT (22H1THE LARGEST OF THESE I3, 12H NUMBERS IS F7.2)
      STOP 77777
```

FORTRAN I

- The first compiler
  - Huge impact on computer science
- Led to an enormous body of theoretical and practical work
- Modern compilers preserve the outlines of FORTRAN I
- Can you name a modern compiler?

How are Languages Implemented?

- Two major strategies:
  - Interpreters run your program
  - Compilers translate your program



How does a compiler work?

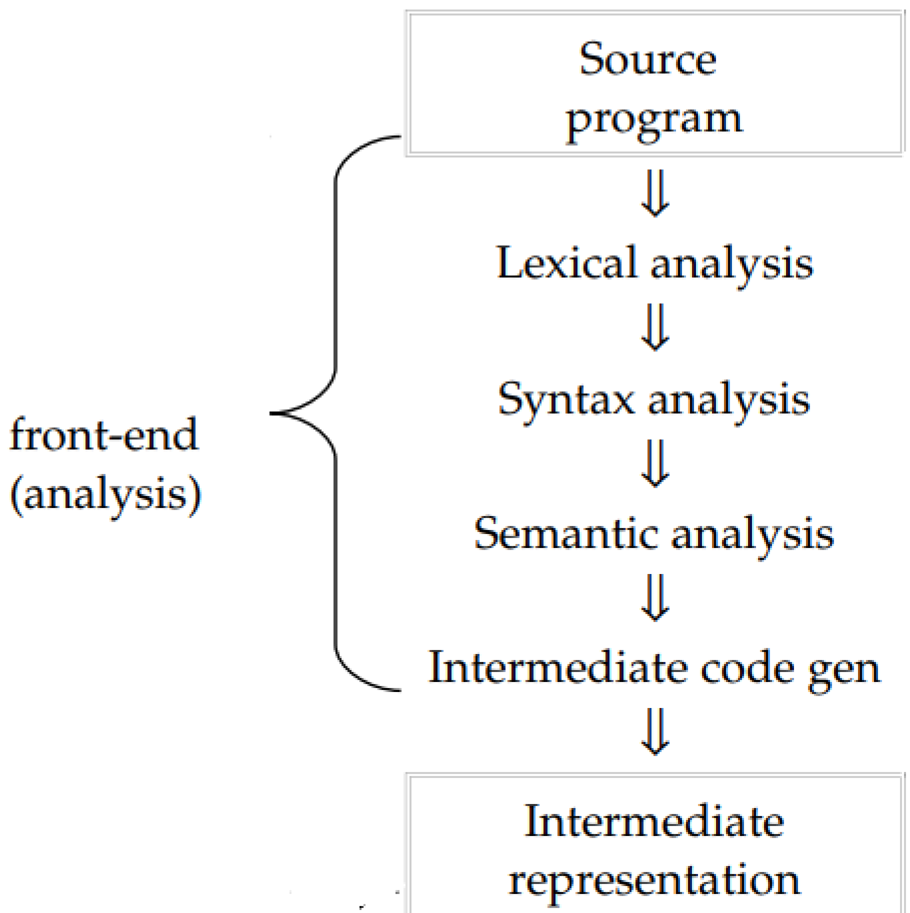
- Two main stages in the compiling process:
  - **Analysis (front end)**
    The analysis stage breaks up the source program into pieces and creates a generic (language independent) intermediate representation of the program.

  - **Synthesis (back end)**
    The synthesis stage constructs the desired target program from the intermediate representation

- Each of the stages is broken down into a set of "phases" that handle different parts of the tasks.

How does a compiler work?

- **Analysis (front end)**

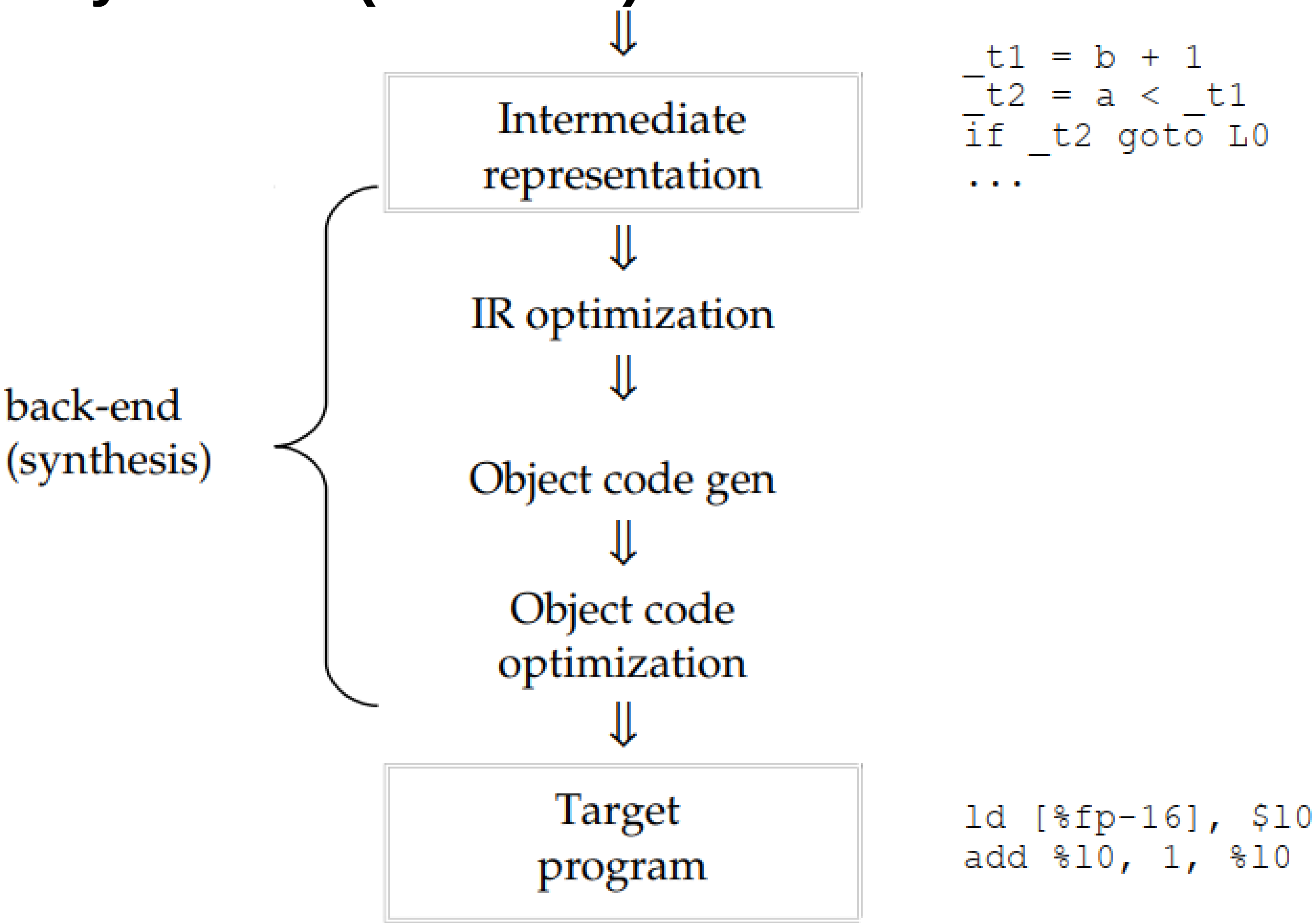



```
if (a >= b+1) {
  a *= 2;
}
...
```

```
_t1 = b + 1
_t2 = a < _t1
if _t2 goto L0
...
```

How does a compiler work?

- **Synthesis (back end)**



1. Analysis (front end)

I. Lexical Analysis

- First step: recognize words.
  - Smallest unit above letters

This is a sentence.

Can be understood by analogy to how humans comprehend English.

More Lexical Analysis

- Lexical analysis is not trivial. Consider:

ist his ase nte nce

I. Lexical Analysis

The stream of characters making up a source program is read from left to right and grouped into tokens, which are sequences of characters that have a collective meaning.

- Examples of tokens are identifiers (user defined names), reserved words, integers, doubles or floats, delimiters, operators, and special symbols.

1. Lexical Analysis (continue)

Example of lexical analysis:

```
int a;
a = a + 2;
```

A lexical analyzer scanning the code fragment above might return:

`int`	`T_INT`	(reserved word)
`a`	`T_IDENTIFIER`	(variable name)
`;`	`T_SPECIAL`	(special symbol with value of **";"**)
`a`	`T_IDENTIFIER`	(variable name)
`=`	`T_OP`	(operator with value of **"="**)
`a`	`T_IDENTIFIER`	(variable name)
`+`	`T_OP`	(operator with value of **"+"**)
`2`	`T_INTCONSTANT`	(integer constant with value of **2**)
`;`	`T_SPECIAL`	(special symbol with value of **";"**)

I. Lexical Analysis (continue)

Example of lexical analysis:

```
int a;
a = a + 2;
```

A lexical analyzer scanning the code fragment above might return:

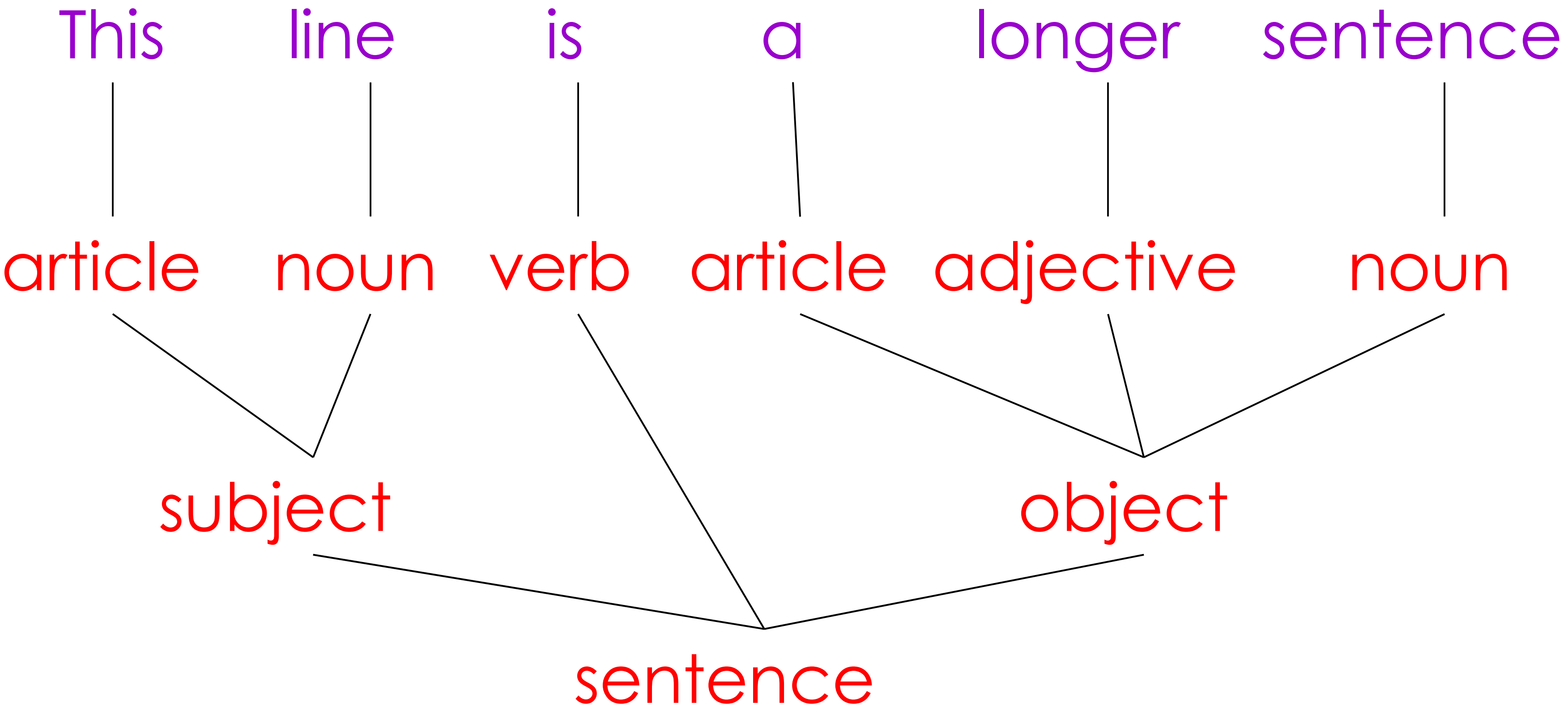
`int`	`T_INT`	(reserved word)
`a`	`T_IDENTIFIER`	(variable name)
`;`	`T_SPECIAL`	(special symbol with value of `";"`)
`a`	`T_IDENTIFIER`	(variable name)
`=`	`T_OP`	(operator with value of `"="`)
`a`	`T_IDENTIFIER`	(variable name)
`+`	`T_OP`	(operator with value of `"+"`)
`2`	`T_INTCONSTANT`	(integer constant with value of `2`)
`;`	`T_SPECIAL`	(special symbol with value of `";"`)

II. Syntax Analysis (Parsing)

- The tokens found during scanning are grouped together using a context free grammar.
- A grammar is a set of rules that define valid structures in the programming language.
- Each token is associated with a specific rule, and grouped together accordingly.

- Parsing = Diagramming Sentences
  - The diagram is a tree
  - Derivation

Diagramming a Sentence

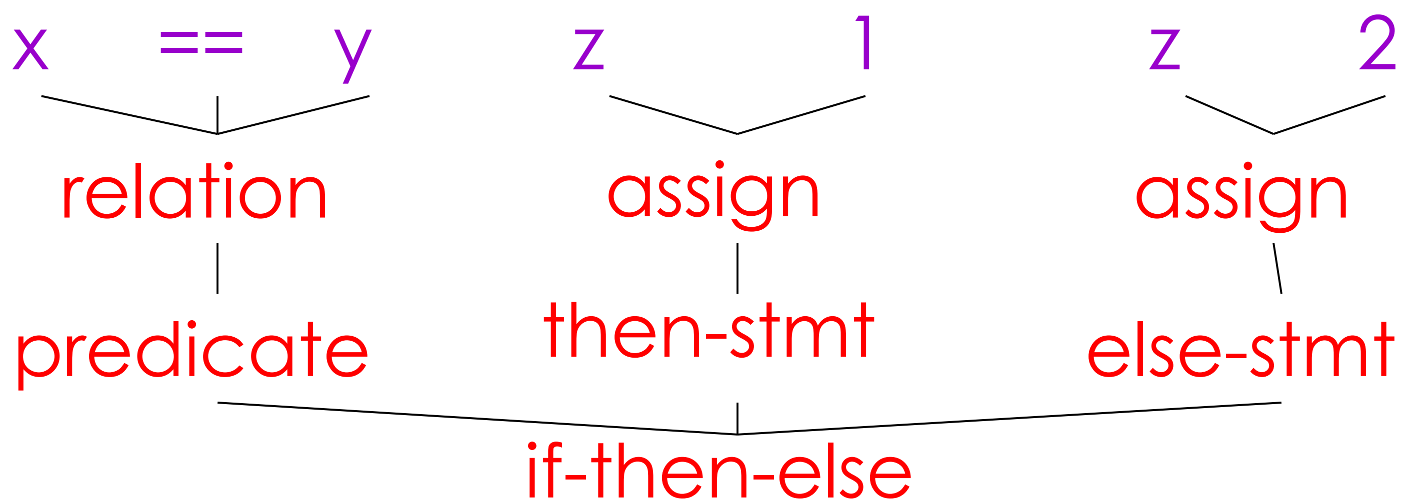


Parsing Programs

- Parsing program expressions is the same
- Consider:

  If x == y then z = 1; else z = 2;

- Diagrammed:

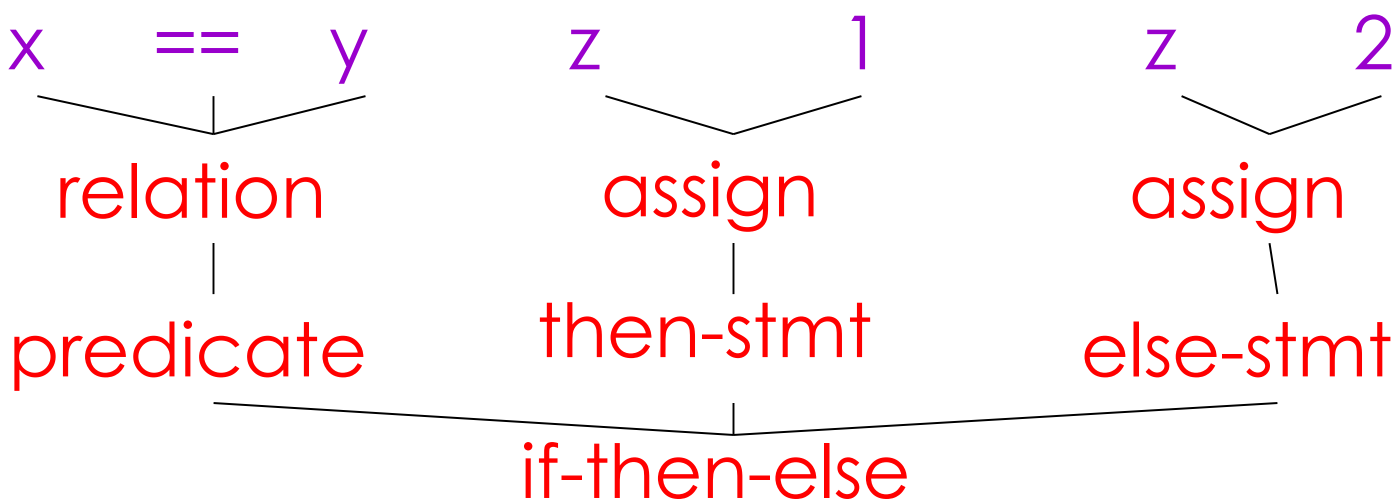


Parsing Programs

- Parsing program expressions is the same
- Consider:

If x == y then z = 1; else z = 2;

- Diagrammed:



Parsing Programs

- To parse a + 2, we would apply the following rules:

```
Expression -> Expression + Expression
           -> Variable + Expression
           -> T_IDENTIFIER + Expression
           -> T_IDENTIFIER + Constant
           -> T_IDENTIFIER + T_INTCONSTANT
```

When we reach a point in the parse where we have only tokens, we have finished.

III. Semantic Analysis

- Once sentence structure is understood, we can try to understand “meaning”
  - But meaning is too hard for compilers
- This phase also checks the parse tree or derivation for semantic errors.
- Semantic analysis is the phase where we detect such things as use of an undeclared variable, a function called with improper arguments, access violations, and incompatible operands and type mismatches, e.g., an array variable added to a function name.

Semantic Analysis in English

- Example:

  Tom said Jerry left his assignment at home.

  What does “his” refer to? Tom or Jerry?

- Even worse:

  Tom said Tom left his assignment at home?

  How many Toms are there?

  Which one left the assignment?

Semantic Analysis in Programming

- Programming languages define strict rules to avoid such ambiguities

- This C++ code prints "4"; the inner definition is used

```
{
    int Jack = 3;
    {
        int Jack = 4;
        cout << Jack;
    }
}
```

More Semantic Analysis

- Compilers perform many semantic checks besides variable bindings

- Example:

  Jack left her homework at home.

- Possible type mismatch between her and Jack
  - If Jack is male

IV. Intermediate Code Generation

- This phase generates intermediate representation of the source program.
- The representation can have a variety of forms, but a common one is called three address code (TAC), which is a lot like a generic assembly language.
- Three address code is a sequence of simple instructions, each of which can have at most three operands.

Intermediate Code Generation(continue)

- Example of intermediate code generation:
  - Source Code:

```
a = b * c + b * d
```

  - Intermediate Code:

```
_t1 = b * c
_t2 = b * d
_t3 = _t1 + _t2
a = _t3
```

2. Synthesis(back end)

I. Intermediate Code Optimization

- Akin to editing
  - Minimize reading time
  - Minimize items the reader must keep in short-term memory
- Automatically modify programs so that they
  - Run faster
  - Use less memory
  - In general, to use or conserve some resource

Intermediate Code Generation(continue)

- Example of code optimization:
  - Intermediate Code:

```
_t1 = b * c
_t2 = _t1 + 0
_t3 = b * c
_t4 = _t2 + _t3
a = _t4
```

  - Optimized Intermediate Code:

```
_t1 = b * c
_t2 = _t1 + _t1
a = _t2
```

II. Object Code Generation

- This phase usually generates machine code or assembly code.
- Memory locations are selected for each variable.
- Instructions are chosen for each operation.
- The three address code is translated into a sequence of assembly or machine language instructions that perform the same tasks

Object Code Generation (continue)

- Example of code generation::
  - Intermediate Code:

```
_t1 = b * c
_t2 = _t1 + _t1
a = _t2
```

  - Intermediate Code:

```
ld [%fp-16], %l1     # load
ld [%fp-20], %l2     # load
smul %l1, %l2, %l3   # mult
add %l3, %l3, %l0    # add
st %l0, [%fp-24]     # store
```

Compilers Today

- The overall structure of almost every compiler adheres to our outline

- The proportions have changed since FORTRAN
  - Early: lexing and parsing most complex/expensive
  - Today: optimization dominates all other phases, lexing and parsing are well understood and cheap

- Compilers are now also found inside libraries

Language Implementations

- Compilers dominate low-level languages
  - C, C++, Go, Rust
- Interpreters dominate high-level languages
  - Python, Ruby
- Some language implementations provide both
  - Java, Javascript, WebAssembly
  - Interpreter + Just in Time (JIT) compiler

Issues

- Compiling is almost this simple, but there are many pitfalls

- Example: How to handle erroneous programs?

- Language design has big impact on compiler
  - Determines what is easy and hard to compile
  - Course theme: many trade-offs in language design

Home Work

- Finite automata review

- Nondeterministic finite automaton(NFA)