CISS445 Lecture 1: Introduction

Yihsiang Liow

January 6, 2020

Table of contents I

1. Course
2. Language paradigm
3. Phases of compilation

Objectives

- New programming paradigm
  - Learn functional (and logic programming)
  - Environments ad closures
  - Patterns of recursion
  - Continuation passing style
- Phases of interpreter / compiler
  - Lexing and parsing
  - Knowledge for writing parsers, interpreters, compilers

Books

- I'll be using my own notes. Some extra references below if you want to study more on your own.
- Programming Language Pragmatics, by Michael L. Scott. Morgan Kaufman Publishers
- Essentials of Programming Languages, by Daniel P. Friedman, Mitchell Wand and Christopher T. Haynes, MIT Press 2001
- Compilers: Principles, Techniques, and Tools (also known as "The Dragon Book"), by Aho, Sethi, and Ullman, Addison-Wesley

Homework/Assignment/etc.

- You may discuss homework/assignments/projects/etc. and their solutions with others
- You must write your own solution
- You may not look at another written solution when you are writing your own
  - You may look at examples from class, textbooks and other similar examples

Software

- Get latest fedora virtual machine (Spring 2018: Fedora 25)
- Install OCAML and SWI-Prolog

```
dnf -y install ocaml
dnf -y install pl
```

Paradigms,allowframebreaks

- Imperative
- Object-oriented
- Applicative/functional
- Logic
- Etc.

Imperative languages I

- Main focus: machine state the set of values stored in memory locations
- Command-driven: Each statement uses current state to compute a new state
- Example languages: C/C++, Pascal, Python, FORTRAN, COBOL

Object-oriented languages I

- Classes are complex data types grouped with methods for creating, examining, and modifying objects.
- Subclasses include (inherit) the objects and methods from superclasses
- Computation is based on objects sending messages (methods applied to arguments) to other objects
- Example languages: Java, C++, Python, Smalltalk

Functional languages I

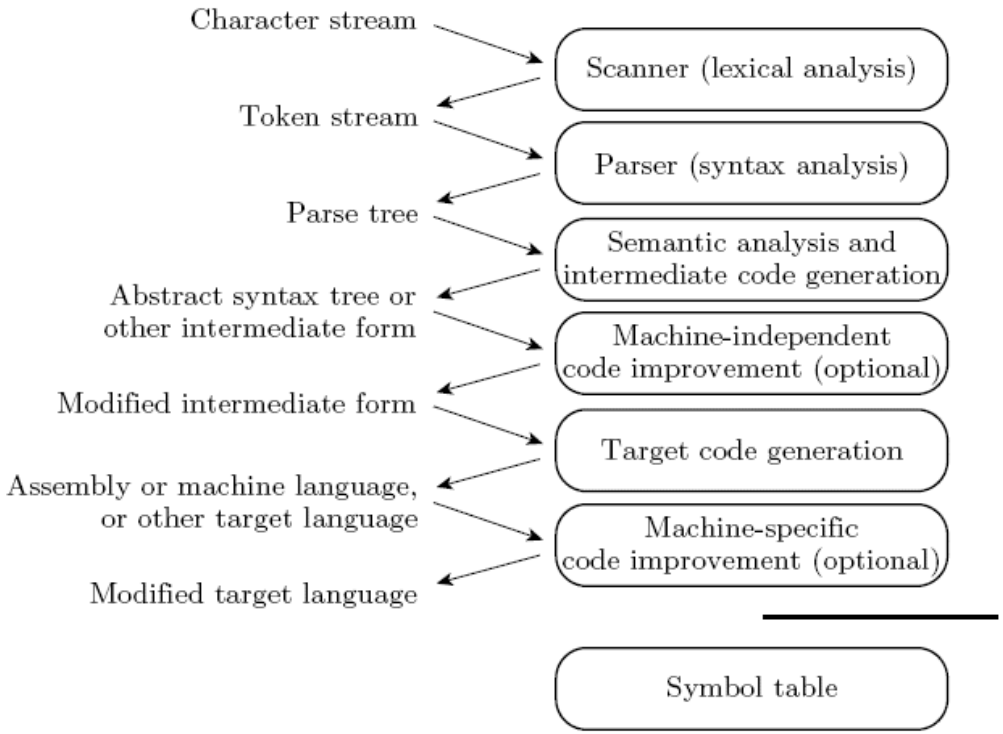
- Functional/applicative languages: Programs as functions that take arguments and return values; arguments and returned values may be functions
- Programming consists of building the function that computes the answer; function application and composition main method of computation
- Example languages: ML, LISP, Scheme, Haskell, Miranda

Logic/constraint-based/rule-based languages I

- Programs as sets of basic rules for decomposing problem
- Computation by deduction: search, unification and backtracking main components
- Example languages: Prolog

Phases of compilation I

- Phases of compilation:



Phases of compilation II

- **Scanning**: Divides the program into meaningful substrings, **lexemes**, which are converted to **tokens**. A token is the smallest meaningful unit of a program.
  - This saves time, since char-by-char processing is slow
  - We can tune the scanner better if its job is simple; it also saves complexity (lots of it) for later stages
  - Can design a parser to take chars instead of tokens as input, but not pretty
  - Scanning is recognition of a **regular language**, e.g., via **DFA** = **deterministic finite state automata**

Phases of compilation III

- **Parsing** is recognition of a context-free language, e.g., via PDA = pushdown automata
  - Parsing discovers the "context free" structure of the program
  - Informally, it finds the structure you can describe with syntax diagrams

Phases of compilation IV

- **Semantic analysis** is the discovery of meaning in the program
  - The compiler actually does what is called STATIC semantic analysis. That's the meaning that can be figured out at compile time
  - Some things (e.g., array subscript out of bounds) can't be figured out until run time. Things like that are part of the program's DYNAMIC semantics

Phases of compilation V

- **Intermediate form (IF)** done after semantic analysis (if the program passes all checks)
  - IFs are often chosen for machine independence, ease of optimization, or compactness (these are somewhat contradictory)
  - They often resemble machine code for some imaginary idealized machine; e.g. a stack machine, or a machine with arbitrarily many registers
  - Many compilers actually move the code through more than one IF

Phases of compilation VI

- **Optimization** takes an intermediate-code program and produces another one that does the same thing faster, or in less space
  - The term is a misnomer; we just improve code
  - The optimization phase is optional
- **Code generation phase** produces assembly language or relocatable machine language

Phases of compilation VII

- Certain **machine-specific optimizations** (use of special instructions or addressing modes, etc.) may be performed during or after **target code generation**
- **Symbol table**: all phases rely on a symbol table that keeps track of all the identifiers in the program and what the compiler knows about them
  - This symbol table may be retained (in some form) for use by a debugger, even after compilation has completed

Example I

GCD Program in Pascal.

```
program gcd(input, output);
var i, j: integer;
begin
    read(i, j);
    while i <> j do
        if i > j then i := i - j
        else j := j - 1;
    writeln(i)
end.
```

Example II

Tokens:

```
program  gcd      (        input    ,        output   )        ;
var      i        ,        j        :        integer  ;        begin
read     (        i        ,        j        )        ;        while
i        <>       j        do       if       i        >        j
then     i        :=       i        -        j        else     j
:=       j        -        i        ;        writeln  (        i
)        end      .
```

Example III

Context-Free Grammar

$$\textit{program} \longrightarrow \texttt{PROGRAM id (id } \textit{more\_ids} \texttt{) ; } \textit{block} \texttt{ .}$$

where

$$\textit{block} \longrightarrow \textit{labels constants types variables subroutines } \texttt{BEGIN } \textit{stmt more\_stmts } \texttt{END}$$

and

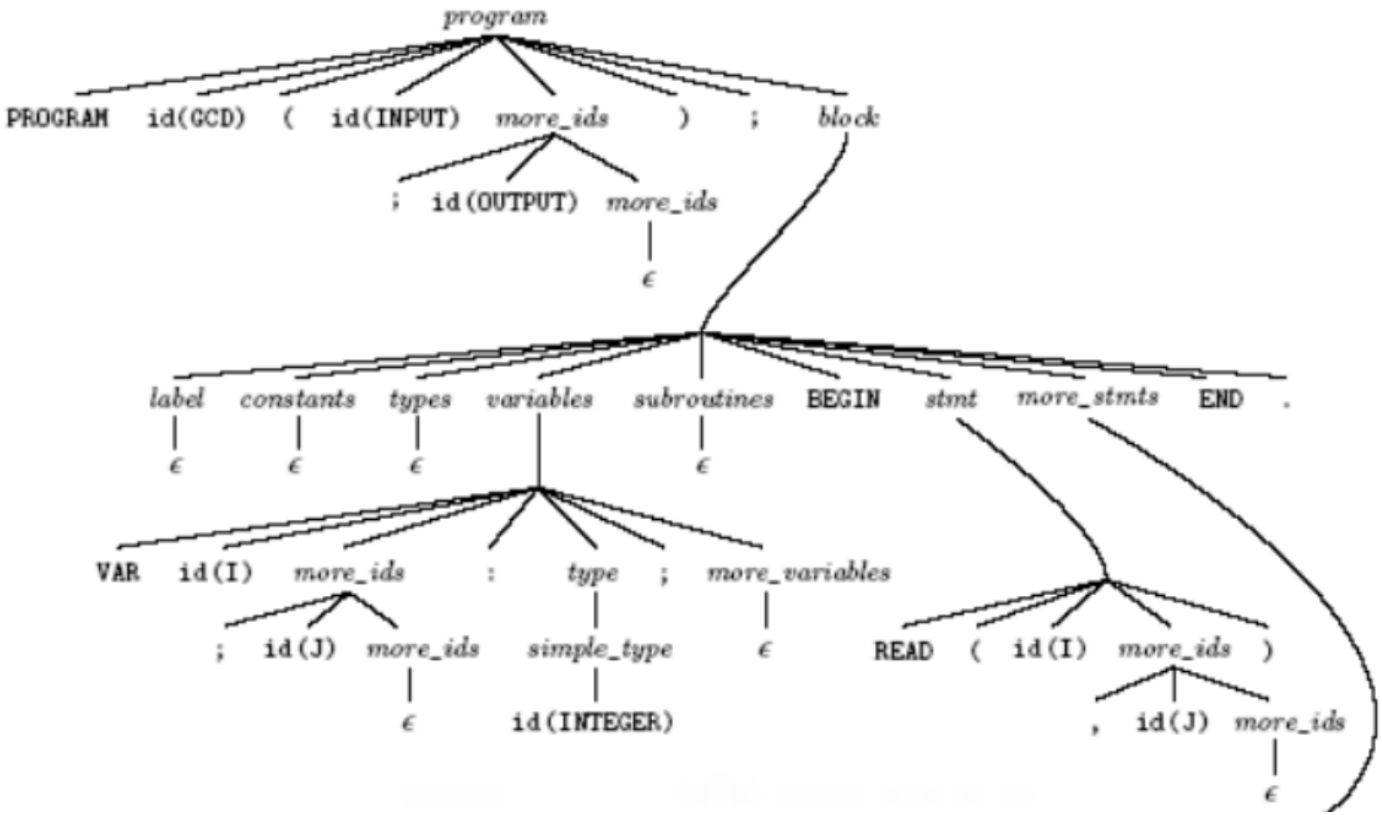
$$\textit{more\_ids} \longrightarrow \texttt{ , id } \textit{more\_ids}$$

or

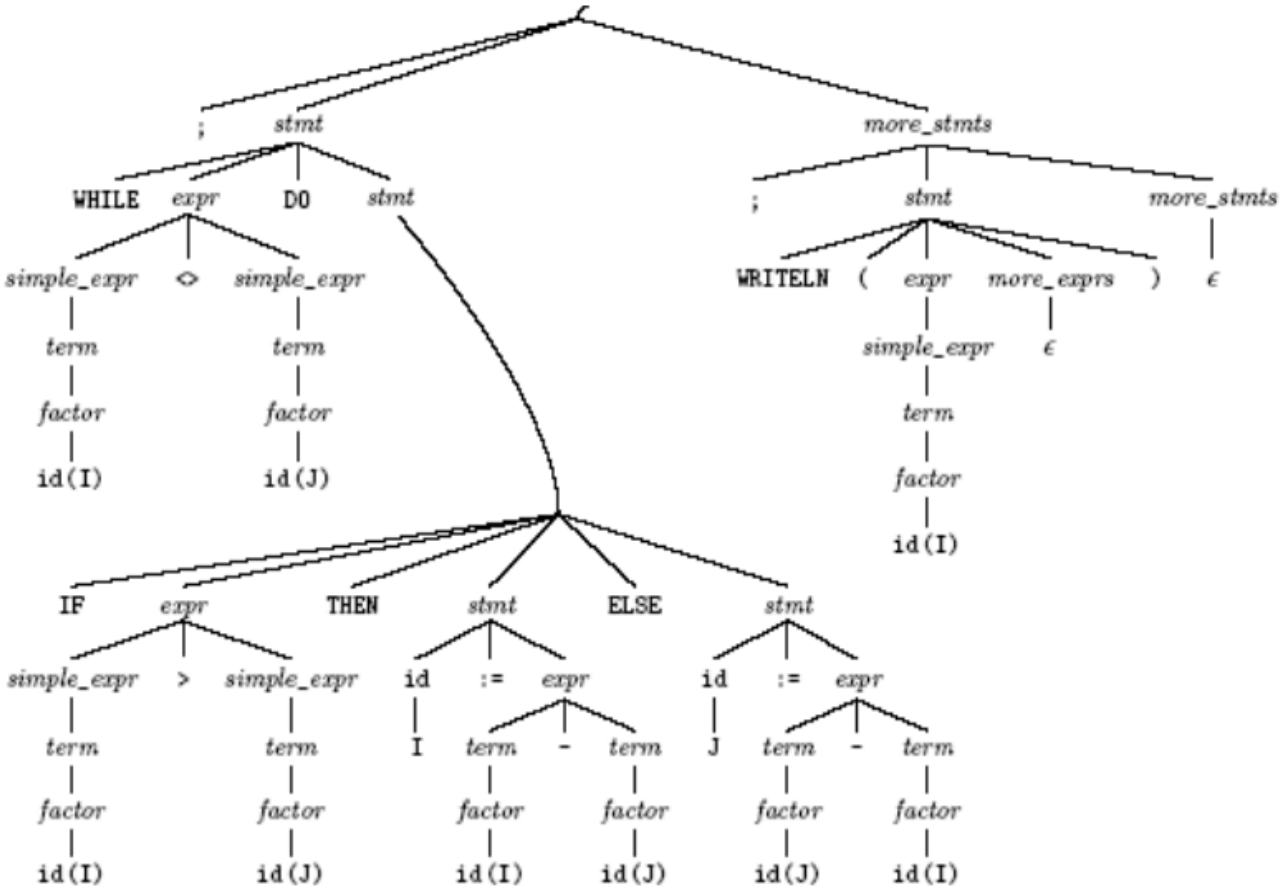
$$\textit{more\_ids} \longrightarrow \epsilon$$

Example IV

Parse tree

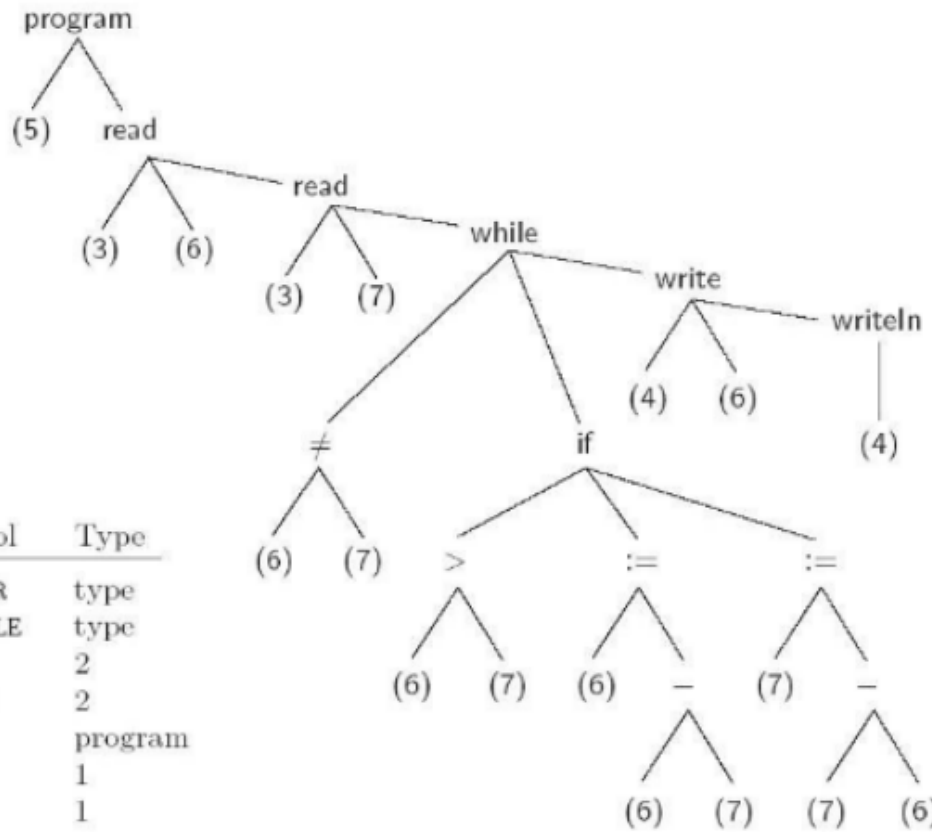


Example V


;
stmt
more_stmts
WHILE
expr
DO
stmt
;
stmt
more_stmts
simple_expr
<>
simple_expr
WRITELN
(
expr
more_exprs
)
ϵ
term
term
simple_expr
ϵ
factor
factor
term
id(I)
id(J)
factor
id(I)
IF
expr
THEN
stmt
ELSE
stmt
simple_expr
>
simple_expr
id
:=
expr
id
:=
expr
term
term
I
term
-
term
J
term
-
term
factor
factor
factor
factor
factor
factor
id(I)
id(J)
id(I)
id(J)
id(J)
id(I)

Example I

Syntax tree and symbol table



Index	Symbol	Type
1	INTEGER	type
2	TEXTFILE	type
3	INPUT	2
4	OUTPUT	2
5	GCD	program
6	I	1
7	J	1