Programming Language Implementation

Translation

Source language programs are translated into *target language* programs:

- **Assembler:** symbolic representation of machine code $\rightarrow$ machine code
- **Compiler:** high(er)-level language $\rightarrow$ low(er)-level language
- **Loader / link editor** translates address references in object code indicated by address tables to actual addresses
- **Macroprocessor / preprocessor** performs macro expansion and code fragment selection by applying rewriting rules

Software simulation — Virtual machines

Create a (low-level) program that acts as a "computer whose machine language is the high-level language".

This **interpreter** also acts as a **virtual machine** implementation.

Most "interpreters" first perform compilation into some internal representation (sometimes exported as **bytecode**).

Stages in Translating a Program

Lexical analysis (Scanner): Breaking a program into primitive components, called **tokens** (identifiers, numbers, keywords, …)

Syntactic analysis (Parsing): Creating a syntax tree of the program.

Symbol table: Storing information about declared objects (identifiers, procedure names, …)

Semantic analysis: Understanding the relationship among the tokens in the program.

Optimization: Rewriting the syntax tree to create a more efficient program.

Code generation: Converting the parsed program into an executable form.

Describing Programs

Syntax — *Shape* of PL constructs

- What are the **tokens** of the language? — **Lexical** syntax, "word level"
- How are programs built from tokens? — Mostly use **Context-Free Grammars** (CFG) or **Backus-Naur-Form** (BNF) to describe **syntax** at the "sentence level"

Semantics — *Meaning* of PL constructs

- Three major approaches to PL semantics:
  - **Axiomatic semantics:** $\{p\}$ `Prog` $\{q\}$
  - **Denotational semantics:** `Prog` *denotes* a *mathematical* function $[\![$`Prog`$]\!]$
  - **Operational semantics:** state transition sequence of an abstract machine
- **"Static semantics":** aspects of program structure that are checked at compile time, but cannot be captured by CFGs ($\rightarrow$ context-sensitive syntax):
  - Scopes of names
  - Typing

Formal Languages, Grammars, Automata

A **formal language** *over an alphabet A* is a subset of A^*.

Formal languages can be *generated* by **grammars**, *recognized* by **automata**.

Phase	*Input Alphabet*	*Output*	*Grammar Type*	*Recognising Automata*	*Generators*
Lexing	Characters	Token **Sequence**	Type 3: Regular	Finite Automata	`lex, flex` `ocamllex` `alex`
Parsing	Tokens	Syntax **Tree**	Type 2: Context-Free	Pushdown Automata	`yacc, bison` `ANTLR, JavaCC` `ocamlyacc,` `happy`

Two formal languages:

- **token language** over character-level alphabet
- **program language** over token alphabet

Token Example: Identifiers in Java

Java 2 Language Spec. 3.8:

```
IdentifierChars:
        JavaLetter
        IdentifierChars JavaLetterOrDigit
```

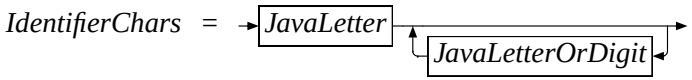
Conventional BNF:

```
IdentifierChars ::=  JavaLetter
                   |  IdentifierChars JavaLetterOrDigit
```

Conventional CFG:

```
IdentifierChars  →  JavaLetter
IdentifierChars  →  IdentifierChars JavaLetterOrDigit
```

"Railroad diagram":



Regular Expression:

$$\mathit{IdentifierChars} = \mathit{JavaLetter} \cdot \mathit{JavaLetterOrDigit}^*$$

BNF in the Textbook

```
Integer  →  Digit  |  Integer Digit
Digit    → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

This is an abbreviation for the following set of CFG rules:

```
Integer  →  Digit
Integer  →  Integer Digit
Digit    → 0
Digit    → 1
           ⋮
Digit    → 9
```

Definition: A **context-free grammar (CFG)** is a tuple (Σ, N, S, ρ) where

- Σ is a set of **terminal symbols**
- N is a set of **nonterminal symbols**
- $S \in N$ is the **start** symbol
- $\rho \subseteq (N \times (N \cup \Sigma)^*)$ is a set of rules;

  a rule (A, ω) is usually written "$A \to \omega$"

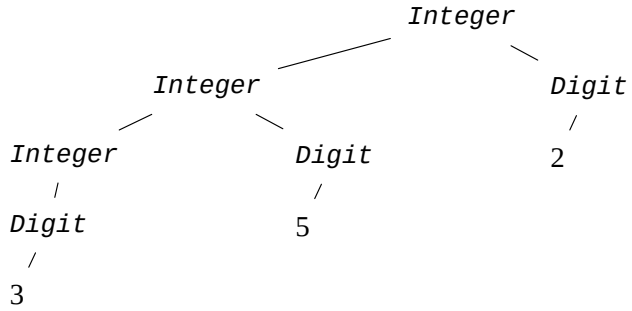
Derivations and Parse Trees

Integer → *Integer Digit* → *Integer Digit Digit*

→ *Digit Digit Digit* → 3 *Digit Digit* → 35 *Digit* → 352

Integer → *Integer Digit* → *Integer* 2

→ *Integer Digit* 2 → *Integer* 52 → *Digit* 52 → 352



Read…

Chapter 2

- Read the UNIX manual pages for `grep` and `egrep`; compare the regular expressions there with those in the textbook.
- Learn what `awk` and `sed` are used for (UNIX texts, manual pages), and what the basic structure of `awk` and `sed` scripts is. Have you ever encountered any problems that you now would solve using `grep` `awk`, and `sed`?