### Introduction

**A. Pnueli**

#### Why Study Compiler Construction?

- **Development of more expressive (and user-friendly) high-level programming languages.**
- **Understanding of many deep issues in programming languages and their execution, e.g. recursion, multi-threading, object-orientation.**
- **Many applications (structured-editorial edition, pretty printers, etc.) use components of a compiler, e.g. error messages and translation.**
- **The study of compilers clarifies many deep issues in programming languages and execution.**

- **Current Trends:**
  - **Expressibility and Maximal Efficiency:** Compilers act as bridges between maximal expressibility and maximal efficiency.

#### Current Trends

- **Branching (Conflicting) Trends:**
  - The compiler should be able to recognize these two (sometimes conflicting) trends.
  - The compiler should enable more efficient execution.

- **Optimization:**
  - Understanding a compiler and its optimization mechanisms enable us to write more efficient programs.
  - Understanding a compiler enables us to write more efficient programs.

- **Current Trends:**
  - **Expressibility and Maximal Efficiency:** Compilers act as bridges between maximal expressibility and maximal efficiency.

#### What is a Compiler?

A. Compiler: A translator from a source to a target program.
A compiler is often applied as a stage within a sequence of transformations:

Language

Preprocessor

Source Program

Compiler

Target Assembly Program

Loader/Link-Editor

Relocatable Machine Code

Assembler

Library

Relocatable Object

Absolute Machine Code

Files

\[ \text{Program} \]

Theory of abstract interpretation and program analysis.

Type theory and its logics.

Formal languages and the theory of parsing.

Semantics of programming languages.

Study of Compilers

Fields and Disciplines that Grew Out of the Study of Compilers

A compiler is often applied as a stage within a sequence of transformations:

Abstract Syntax Trees

\[ \begin{split} & C * B + V \\ & \text{Term} \end{split} \]
Analysiscanbepartitionedintothreephases:

1. **Linear (Lexical) Analysis.** Stream of characters is read left-to-right and partitioned into tokens.

2. **Hierarchical (Syntax) Analysis.** Tokens are grouped hierarchically into nested collections.

3. **Semantic Analysis.** Checking global consistency. Often does not comply with hierarchical structure. Type checking is an instance of such analysis.

Illustrate on a Statement:

```
+ id1 + id2 + id3 * temp1
```

```
+ id1 + id2 + id3 * temp1
```

Processing Continued (2/3):

```
0 + * + id2 + id3 * temp1
```

```
0 + * + id2 + id3 * temp1
```

Semantic Analyzer

Intermediate Code Generator

Lexical Analyzer

Function Call Graph

Phases of a Compiler

Lecture 1: Introduction

A. Pnueli

Honors Compilers, NYU, Fall, 2009
A context-free grammar has four components:

1. A set of terminals, known as terminal symbols.
2. A set of nonterminals (corresponding to categorical concepts).
3. A set of productions of the form $A \rightarrow B_1 \ldots B_k$, where $A$ is a non-terminal, and each $B_i$ is a symbol.
4. A designation of one of the nonterminals as the start symbol.

We often combine the following two rules:

$A \rightarrow B_1 B_2 \ldots B_k$ and $A \rightarrow C_1 C_2 \ldots C_m$ into

$A \rightarrow B_1 B_2 \ldots B_k C_1 C_2 \ldots C_m$.

Example:

List of digits separated by + or -.

A context-free grammar defines a language.

A grammar derives strings by beginning with the start symbol, and repeatedly replacing a non-terminal symbol by the right side of a production for that symbol.

Many compilers produce symbol assembly code which is later translated into relocatable code.

For example, the assembler code corresponding to the source statement

$q := p + q$,

could be:

```assembly
    MOV R1, R1 + 0  // Load the address of p into R1.
    ADD R1, R1 + 0  // Add q to the address of p.
    ADD R1, R1 + 0  // Add q to the address of p.
```

The translation could be:

```assembly
    MOV R1, R1 + 0  // Load the address of p into R1.
    ADD R1, R1 + 0  // Add q to the address of p.
    ADD R1, R1 + 0  // Add q to the address of p.
```

This mode of definition is generalized to the notion of a context-free grammar.
Example of a Derivation

Given the grammar

\[ L \Rightarrow L + D \]

\[ D \Rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 \]

We can use it to derive the string \( 9+2 \) as follows:

\[ \begin{align*}
L &\Rightarrow L + D \\
L &\Rightarrow D \\
D &\Rightarrow 9 \\
D &\Rightarrow 5 \\
D &\Rightarrow + \\
D &\Rightarrow 2 \\
D &\Rightarrow + \\
D &\Rightarrow 9 \\
D &\Rightarrow 5 + 2
\end{align*} \]

The process inverse to derivation is recognition or parsing.

Ambiguous Grammars

A grammar is ambiguous if it can produce two different parse trees for the same string.

For example, following is the parse tree of the derivation of the string \( 9+2 \):

Parse Trees

The history of derivation of a string by a grammar can be represented by a parse tree.

Honors Compilers, NYU, Fall, 2009
Consider the extended grammar

\[ E \rightarrow E + T | T \]
\[ T \rightarrow T \cdot D | D \]
\[ D \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 \]

We can interpret it as capturing the following definitions:

- An expression \( E \) is a sequence of terms separated by operators \(+\) or \(\cdot\).
- A term \( T \) is a sequence of digits separated by the operators \(+\) or \(\cdot\).
- A grammar such as:

\[
\{ \{ q := p \} = q \} = q \qquad \text{as} \qquad q := p = q = q := q
\]

### Capturing Associativity

A grammar such as:

\[ E \rightarrow E + T | T \]
\[ T \rightarrow T \cdot D | D \]
\[ D \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 \]

is called right-recursive and captures right associativity. It will parse the string:

\[
1 + 2 + 3 + 4
\]

In contrast, the grammar:

\[ T \rightarrow T \cdot D | D \]
\[ D \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 \]

is called left-recursive and captures left associativity. It will parse the string:

\[
a := b := c := d := 5
\]

We can interpret it as capturing the following definition:

\[
\{ \{ a \cdot b \} \cdot c \} \cdot d = a
\]

We can interpret it as capturing the following definition:

\[
\{ a \cdot \{ b + \{ c + \{ d + \{ e + \{ f + \{ g + \{ h + \{ i + \{ j + \{ k + \{ l + \{ m + \{ n + \{ o \} \} \} \} \} \} \} \} \} \} \} \} \} \} \} \} \} \}
\]