What is a Compiler?

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

Context of a Compiler

Why Study Compiler Construction?

Additional recommended readings will be listed in the course's web page.


http://www.cs.nyu.edu/courses/spring07/G22.3130-001/index.htm

Please register at the course's class list:

http://www.cs.nyu.edu/mailman/listinfo/g22.31330.001.sp07

Copies of presentations and lecture notes will be available at

Honors Compilers

Amir Pnueli

Honors Compilers, NYU, Spring, 2007

Lecture 1: Introduction

A. Pnueli

Why Study Compiler Construction?

We do not expect many of you to become compiler builders.

However,

• Understanding a compiler and its optimization mechanisms enable us to write more efficient programs.
• Understanding a compiler and its optimization mechanisms enable us to write more efficient programs.
• Many applications (such as debuggers, pretty-printers, etc.) use components of a compiler's error analysis and translation.
• The study of compilers clarifies many deep issues in programming languages and their execution, e.g., recursion, multi-threading. It may help you design your own mini-language.

Basic issues in programming languages and their execution, e.g., recursion, multi-threading. It may help you design your own mini-language.

Understanding a compiler and its optimization mechanisms enable us to write more efficient programs.

Introduction

What is a Compiler?

Compiler: A translator from a source to a target program.
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.

Type checking is an instance of such analysis.

Collections:
- Hierarchical (Syntax) Analysis: Tokens are grouped hierarchically into nested collections.
- Linear (Lexical) Analysis: Stream of characters is read left-to-right and

Semantic Analysis can be partitioned into three phases:

- Checking global consistency. Often does not comply with hierarchical structure.
- Type checking is an instance of such analysis.

Phases of a Compiler:
- Source Program
- Intermediate Code Generator
- Syntax Analyzer
- Semantic Analyzer
- Symbol Table Manager
- Code Optimizer
- Code Generator
- Error Handler

Honors Compilers, NYU, Spring, 2007
Lecture 1: Introduction

A. Pnueli

Illustrate a Statement

Syntax Analyzer

Symbol Table

1. position

2. initial

3. rate

: =

+ id 1 id 2 id 3 60

Lexical Analyzer

position

: =

initial + rate 60

Honors Compilers, NYU, Spring, 2007

Lecture 1:

Processing Continued (2/3)

MOVF d 3, R 2
MULF #60.0, R 2
MOVF d 2, R 1
ADDF R 2, R 1
MOVFR R 1, d 1

temp 1 := inttoreal(60)
temp 2 := d 3 * temp 1
temp 3 := d 2 + temp 2
d 1 := temp 3

Code Generator

Code Optimizer

Code Generator

temp 1 := d 3 * 60.0
d 1 := d 2 + temp 1

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

Assemblers

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

Assemblers

For example, the assembler code corresponding to the source statement

\[ b = a + 2 \]

could be

\[ \text{MOV a, R1} \]
\[ \text{ADD #2, R1} \]
\[ \text{MOVR1, b} \]

The first pass may decide to allocate \( a \) to address \( D + 0 \) and \( b \) to address \( D + 4 \).

The translation could be:

\[ \text{MOV R1, b} \]
\[ \text{ADD #2, R1} \]
\[ \text{MOVR1, R1} \]

A typical assembler proceeds in two passes. The first pass determines addresses and places constants in their right addresses.

The second pass translates the code, replacing references to variables by their addresses or identifiers (relative to the beginning of the program or to the beginning of the data area). Picking these addresses in the symbol table.

Processing Continued (3/3)
A.Pnueli

Syntax Definition

Many definitions of a syntax of a sentence in logic or in a programming language have the following form:

- A number or an identifier is an expression.
- If $E_1$ and $E_2$ are expressions, then so are:
  - $E_1 + E_2$,
  - $E_1 - E_2$,
  - $E_1 \cdot E_2$,
  - $E_1 / E_2$,
  - $E_1^2$.
- A number or an identifier is an expression.
- A set of productions of the form $V \rightarrow V_1 V_2 \ldots V_n$ where $V$ is a non-terminal, and
  - $V$ is a set of non-terminal symbols (corresponding to categorical concepts).
  - $V_1, V_2, \ldots, V_n$ are terminal symbols.
  - A set of tokens known as terminal symbols.
- A context-free grammar has four components:
  1. A set of terminals, known as terminal symbols.
  2. A set of non-terminals (corresponding to categorical concepts).
  3. A set of productions of the form $V \rightarrow V_1 V_2 \ldots V_n$ where $V$ is a non-terminal, and
  - $V$ is a set of non-terminal symbols (corresponding to categorical concepts).
  - $V_1, V_2, \ldots, V_n$ are terminal symbols.
  - A number or an identifier is an expression.
- The process inverse to derivation is recognition or parsing.

The importance of grammars is not only in their ability to distinguish between acceptable and unacceptble strings. Not less important is the hierarchical grouping they induce on the strings through the parse trees.

For example, following is the parse tree of the derivation of the string $9 + 5 + 2$ by the grammar for $\text{list}$.

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

Example of a Derivation

Given the grammar:

$$L = L + D | \{z\}$$
$$D = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$$
$$L = L + D$$
$$D = \{z\} + D$$

we can use it to derive the string $9 + 5 + 2$ as follows:

- $L = L + D \mid \{z\}$
- $L = 9 \mid \{z\} + D \mid \{z\}$
- $L = 9 + D \mid \{z\}$
- $L = 9 + 5 \mid \{z\}$
- $L = 9 + 5 + 2 \mid \{z\}$
- $L = 9 + 5 + 2$
A grammar is ambiguous if it can produce two different parse trees for the same string.

The grammar for list was unambiguous. On the other hand, the following grammar for the same language is ambiguous:

\[
\begin{align*}
E & \rightarrow E + D \\
D & \rightarrow T \\
T & \rightarrow D \mid (E)
\end{align*}
\]

Among these two parse trees, only the right provides the correct arithmetical grouping.

A grammar that correctly captures the operators precedence is given by:

\[
\begin{align*}
E & \rightarrow E + T \\
T & \rightarrow A \mid T \cdot A
\end{align*}
\]

A grammar that correctly captures the operator precedence is given by:

\[
\begin{align*}
E & \rightarrow E + T \\
T & \rightarrow A \mid T \\
A & \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{align*}
\]

A grammar that correctly captures the associativity is given by:

\[
\begin{align*}
E & \rightarrow E + T \\
T & \rightarrow A \mid T \cdot A
\end{align*}
\]

A grammar that correctly captures the associativity is given by:

\[
\begin{align*}
E & \rightarrow E + T \\
T & \rightarrow A \mid T
\end{align*}
\]

A grammar that correctly captures the associativity is given by:

\[
\begin{align*}
E & \rightarrow E + T \\
T & \rightarrow A \mid T
\end{align*}
\]