Honors Compilers, NYU, Spring, 2007

Mondays, Wednesdays 2:00-3:15 PM

Amir Pnueli

Honors Compilers

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


http://www.cs.nyu.edu/email/index.html

Please register at the course's class list:

http://www.cs.nyu.edu/mailman/listinfo/g22-3130-001-sp07

Copies of presentations and lecture notes will be available at:

http://www.cs.nyu.edu/department/computer-science/courses/ spring07/G22.3130-001/index.html
What is a Compiler?

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

What are Error Messages, Warning Messages, and Documentation?
Why Study Compiler Construction?

We do not expect many of you to become compiler builders. However, many applications (structure-sensitive editors, pretty printers, etc.) use components of a compiler, e.g., *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.
- Understanding a compiler and its optimization mechanisms enable us to write more efficient programs.

For example, the optimization —

\[
\begin{align*}
A &:= B/C; \\
\text{for } i = 1 \text{ to } 100000 \text{ do} \\
&\quad \text{skip}
\end{align*}
\]
A compiler is often applied as a stage within a sequence of transformations:
Lecture 1: Introduction

A. Pnueli

Analysis of the Source Program

Analysis can be partitioned into three phases:

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.

\[ A * B + C \]
Analysis of the Source Program

Analyses can be partitioned into three phases:

- Linear (lexical) analysis: Stream of characters is read left-to-right and

- Hierarchical (syntactic) analysis: Tokens are grouped hierarchically into nested collections.

- Semantic analysis: Checking global consistency. Often does not comply with

\[
\begin{array}{c}
\text{Term} \\
C * B + A
\end{array}
\]
Analyses of the Source Program

A. Pnueli
Honors Compilers, NYU, Spring, 2007

Analyses can be partitioned into three phases:

- **Semantic Analysis**: Checking global consistency. Often does not comply with hierarchical structure. Type checking is an instance of such analysis.
- **Hierarchical (Syntactic) Analysis**: Tokens are grouped hierarchically into nested collections. Tokens are grouped hierarchically into nested collections.
- **Linear (Lexical) Analysis**: Stream of characters is read left-to-right and partitioned into tokens.

A + B * C

Expression

Term

Term

C  B  A

Expression

Semantic Analysis

Hierarchical Analysis

Linear Analysis
Phases of a Compiler

- Source Program
- Lexical Analyzer
- Syntax Analyzer
- Semantic Analyzer
- Code Generator
- Intermediate Code Generator
- Error Handler
- Code Optimizer
- Manager
- Symbol Table
- Target Program
Honors Compilers, NYU, Spring, 2007

Illustrate on a Statement

A. Pnueli
Processing Continued (2/3)

Semantic Analyzer

Intermediate Code Generator

A. Pnueli

Honors Compilers, NYU, Spring, 2007
A. Pnueli

Lecture 1: Introduction

Processing Continued (3/3):

\[
\begin{align*}
\text{temp1} & := \text{inttoreal}(60) \\
\text{temp2} & := \text{id3} \times \text{temp1} \\
\text{temp3} & := \text{id2} + \text{temp2} \\
\text{id1} & := \text{temp3}
\end{align*}
\]

Code Optimizer

\[
\begin{align*}
\text{temp1} & := \text{id3} \times 60.0 \\
\text{id1} & := \text{id2} + \text{temp1}
\end{align*}
\]

Code Generator

Honors Compilers, NYU, Spring, 2007

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

A typical assembler proceeds in two passes:

1. The first pass may decide to allocate variables to addresses. For example, the assembler code corresponding to the source statement:

   \[ a := b + 2 \]

   Could be:

   - First pass:
     - MOV R1, a
     - ADD #2, R1
     - MOVR1, b
   - Second pass:
     - MOV R1, a
     - ADD #2, R1
     - MOVR1, b

   The translation could be:

   \[ 2 \quad 1 \quad 0 \quad 04 \]
   \[ 3 \quad 1 \quad 2 \quad 02 \]
   \[ 1 \quad 1 \quad 0 \quad 00 \]

   \[ \text{Store from R1} \quad + \quad \text{Add into R1} \quad + \quad \text{Load into R1} \]

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

   The first pass determines addresses of identifiers relative to the beginning of the program or to the beginning of the data area, placing these addresses in the symbol table.

   The second pass translates the code, replacing references to variables by their addresses and placing constants in their right addresses.

Assemblers
A.Pnueli

Syntax Definition

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

```
E₁, E₂, E₁ + E₂, E₁ * E₂, E₁ - E₂
```

This mode of definition is generalized to the notion of a context-free grammar. What is characteristic of such definitions is that they start from some atomic constructs (e.g., number, identifier), introduce one or more categorical constructs (e.g., expression), and then provide rules by which constructs can be combined to yield further instances of constructs. (e.g. expression), and then provide rules by which constructs can be combined to yield further instances of constructs.

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

```
E₁, E₂, E₁ + E₂, E₁ * E₂, E₁ - E₂
```

• If $E₁$ and $E₂$ are expressions, then so are $E₁ + E₂$, $E₁ * E₂$, $E₁ - E₂$.

• A number or an identifier is an expression.
A context-free grammar has four components:

1. A set of terminals, known as terminals.
2. A set of non-terminals (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 $B_i$ are terminals or non-terminals.
4. A designation of one of the non-terminals as the start symbol.

In a context-free grammar, any symbol $B_i$ can be replaced by the right side of a production for that terminal, and repeatedly replacing a non-terminal symbol by the right side of a production for that terminal.

Example: List of digits separated by + or –.

$$A \rightarrow \text{list} \mid \text{digit} \mid \text{list} \mid \text{digit} \mid \text{digit} \mid \text{digit} \mid \text{digit} \mid \text{digit} \mid \text{digit} \mid \text{digit} \mid \text{digit}$$
The process inverse to derivation is recognition or parsing.

\[
\begin{align*}
\overbrace{2 + 5 - 6} & \iff \overbrace{a + 5 - 6} \iff a + \overbrace{a - 6} \\
& \iff a + a - \overbrace{a} \iff a + \overbrace{a - 7} \iff \overbrace{a - 7} \\
& \iff I
\end{align*}
\]

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

\[
\begin{array}{cccccccccccc}
6 & 8 & 7 & 6 & 4 & 3 & 2 & 1 & 0 & \leftarrow & D \\
\hline
D & D - 7 & D + 7 & I
\end{array}
\]

Given the grammar

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

The importance of grammars is not only in their ability to distinguish between acceptable and unacceptable 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 list.

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

**Parse Trees**
A grammar is ambiguous if it can produce two different parse trees for the same string. Among these two parse trees, only the right provides the correct arithmetical grouping. From now on, we will restrict our attention to unambiguous grammars.

The grammar for the same language is ambiguous:

\[
2 + 9 - 6 + 5 - 7 + 6 - 4 + 3 - 2 + 0 \to 5 + 2 + 9
\]

The grammar for \textit{list} was unambiguous. On the other hand, the following grammar is ambiguous:

\[
A \text{ grammar is ambiguous if it can produce two different parse trees for the same string.}
\]
Capturing Operator Precedence

Consider the language of parenthesis-free arithmetic expressions over digits and the two operations $+$ and $\times$. A possible grammar for this language is

$$L = E \mid T \mid \{z\} \mid E \cdot \{z\} \mid E + \{z\}$$

Unfortunately, parsing the string $2 + 4 \cdot 3$ according to this grammar yields the grouping $2 + 4 3$ instead of the correct grouping $2 (4 + 3)$.

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

$$L = E \mid T \mid \{z\} \mid E \cdot \{z\} \mid E + \{z\}$$

Parsing the string $2 + 4 \cdot 3$ according to this grammar yields $2 + 4 \cdot 3 = 14$ instead of the correct value 18.

Consider the language of parenthesis-free arithmetic expressions over digits and the two operators + and $\times$. A possible grammar for this language is

$$L = E \mid T \mid \{z\} \mid E \cdot \{z\} \mid E + \{z\}$$

Parsing the string $2 + 4 \cdot 3$ according to this grammar yields $2 + 4 3 = 18$ instead of the correct value 14.

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

$$L = E \mid T \mid \{z\} \mid E \cdot \{z\} \mid E + \{z\}$$

Parsing the string $2 + 4 \cdot 3$ according to this grammar yields $2 + 4 3 = 14$ instead of the correct value 18.

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

$$L = E \mid T \mid \{z\} \mid E \cdot \{z\} \mid E + \{z\}$$

Parsing the string $2 + 4 \cdot 3$ according to this grammar yields $2 + 4 3 = 14$ instead of the correct value 18.
Consider the extended grammar $E$:

- $E = T_0$ or $E + T_1$, or $E T_3$, or $T_4 D$, or $T_5$, or $T_6 7$, or $T_7 8$, or $T_8 9$

We can interpret it as capturing the following definitions:

- An expression $(E)$ is a sequence of terms $(T_1)$ separated by the operators $+$ or $-$.
- A term $(T_1)$ is a sequence of digits $(D)$ separated by the operators $+$ or $-$.

Consider the extended grammar.

Easy Reading of Grammars
A grammar such as

\[
\{\{p = q \mid q = v \mid v\}\} =: A
\]

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

\[
A =: p =: q =: v =: a
\]

In contrast, the grammar

\[
\{\{p = q \mid q = v \mid v\}\} =: A
\]

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

\[
A =: p =: q =: v =: a
\]