Honors Compilers, NYU, Fall, 2009

Additional recommended readings are listed in the course's web page:


http://www.cs.nyu.edu/classes/g22-310-001/fall09/

Please register at the course's class list:

http://www.cs.nyu.edu/mailman/listinfo/g22-310-001-fa09

Copies of presentations and lecture notes will be available at:

http://www.cs.nyu.edu/courses/fall09/G22-310-001/index.html

Tuesdays, Thursdays 3:30-4:45 PM

Amir Pnueli

Honors
Compilers
What is a Compiler?

Compiler: A translator from a source to a target program.
Why Study Compiler Construction?

A.Pnueli

Wedonotexpectmanyofyoutobecomecompilerbuilders. However, Manyapplications(structure-sensitiveeditors,prettyprinters,etc.)use componentsofacompiler,e.g. analysis and translation. Thestudyofcompilersclariesmanydeepissuesinprogramminglanguages andtheirexecution,e.g. recursion, multi-threading, object-orientation. It maybe that thestudyofcompilersclarifiesmanydeeperissuesinprogramminglanguages andtheirexecution,e.g. recursion, multi-threading, object-orientation. It may helpyoudeisgnyourownminilanguage. Theunderlyingcomponentsofacompiler,e.g. analyis andtranslation. Manyapplications(structure-sensitivesystems,prettyprinters,etc.)use

For example, the optimization —

moreefficientprograms.

Understandingacompileranditsoptimizationmechanismsenableuswritenew

Translation.


UnderlyingcompilersconstructionaremanyC5seminalconcepts,suchas...

...
The compiler should be able to reconcile these two (sometimes conflicting) trends:

- Enable more efficient execution.
- Development of more advanced (and parallel) architectures that
  programming languages.
- Development of more expressive (and user-friendly) high-level

Current trends:

**Expressibility and Maximal Efficiency**
Lecture 1: Introduction

Fields and Disciplines that Grew Out of the Study of Compilers

- Theory of abstract interpretation and program analysis.
- Type theory and its logics.
- Formal languages and the theory of parsing.
- Semantics of programming languages.

Honors Compilers, NYU, Fall 2009
A compiler is often applied as a stage within a sequence of transformations:

**Context of a Compiler**

Analysis of the Source Program

A. Pnueli

Analysis can be partitioned into three phases:

1. Linear (lexical) Analysis. Stream of characters is read left-to-right and

\[ C \ast B \ast A \]

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

3. Semantic Analysis. Checking global consistency. Often does not comply with

\[ A \ast \rightarrow B \ast \rightarrow C \ast \rightarrow \]

Honors Compilers, NYU, Fall, 2009

A. Pnueli
Analysis can be partitioned into three phases:

- **Linear (lexical) analysis**. Stream of characters is read left-to-right and
  tokens are grouped hierarchically into nested collections.

- **Hierarchical (syntactic) analysis**. Tokens are grouped hierarchically into nested
terminals.

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

```latex
\text{Term} \quad C \quad * \quad B \quad + \quad A
```

Analyses of the Source Program
Analysis of the Source Program

Analyses can be partitioned into three phases:

1. Linear (lexical) Analysis. Stream of characters is read left-to-right and
   tokens are grouped hierarchically into nested
   collections.

2. Hierarchical (syntax) Analysis. Tokens are grouped hierarchically into nested
   partitions into tokens.

3. Semantic Analysis. Checking global consistency. Often does not comply with
   the hierarchical structure. Type checking is an instance of such analysis.
Lecture 1: Introduction

A. Pnueli

Phases of a Compiler

Error Handler

Source Program

Target Program

Code Generator

Code Optimizer

Intermediate Code Generator

Semantic Analyzer

Syntactic Analyzer

Lexical Analyzer

Symbol Table Manager

Honors Compilers, NYU, Fall, 2009
Lecture 1
Introduction

Syntax Analyzer

Position = Initial + Rate * 60

Lexical Analyzer

Position

Illustrate on a Statement

A. Pnueli

Honors Compilers, NYU, Fall, 2009
Processing Continued (2/3)

Semantic Analyzer

Intermediate Code Generator

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

A typical assembler proceeds in two passes. The first pass determines relocations 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.

The translation could be:

```assembly
MOV R1, a
ADD #2, R1
MOV R1, b
```

For example, the assembler code corresponding to the source statement:

\[ b := a + 2 \]

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

The translation could be:

```
2 1 0 04 +
3 1 2 02 +
4 1 0 00 +
```

Honors Compilers, NYU, Fall, 2009

Assemblers
A number or an identifier is an expression.

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

\[ E_1, E_2, E_1 + E_2, E_1 - E_2, E_1 * E_2, (E_1), \]

This mode of definition is generalized to the notion of a context-free grammar (or simply grammar).

What is characteristic of such inductive definitions is that they start from some atomic constructs (e.g., number, expression) and then provide rules by which constructs can be combined to yield further instances of constructs.
A context-free grammar has four components:

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

Example: list of digits separated by + or -.

```
list \rightarrow list + digit | list - digit | digit
```

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 non-terminal and symbol. A grammar is the language defined by the grammar.
The process inverse to derivation is recognition or parsing.

\[
\begin{align*}
\hat{z} + 5 - 6 & \iff \overline{a} + \overline{z} - 6 \iff a + \overline{a} - 6 \\
\iff a + a - \overline{a} & \iff a + \overline{a} - \overline{7} \iff \overline{a} + \overline{7} \iff \overline{7}
\end{align*}
\]

Given the grammar:

\[
\begin{align*}
6 & \quad 8 & \quad 7 & \quad 6 & \quad 5 & \quad 4 & \quad 3 & \quad 2 & \quad 1 & \quad 0 \iff D \\
D & \quad D - T & \quad D + T \iff T
\end{align*}
\]

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 unacceptible 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 \( 9 - 5 + 2 \).

The history of derivation of a string by a grammar can be represented by a parse tree.
A grammar is ambiguous if it can produce two different parse trees for the same string. Among these two parse trees, only the right one provides the correct arithmetic result. From now on, we will restrict our attention to unambiguous grammars.

A grammar for the same language is ambiguous:

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

A grammar is ambiguous if it can produce two different parse trees for the same string.
Consider the language of parenthesis-free arithmetic expressions over digits

\[ E \rightarrow E + D \\
\rightarrow E \times D \\
\rightarrow D \\
\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \]

Unfortunately, parsing the string \( 2 + 4 \times 3 \) according to this grammar, yields the grouping \( 2 + 4 \times 3 \) instead of \( 2 + (4 \times 3) \). This grouping implies evaluation of the string to the value 18, instead of the correct value 14. A grammar that correctly captures the operator precedence is given by:

\[ E \rightarrow E + T \\
\rightarrow E \times T \\
\rightarrow T \\
\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \]

Parsing the string \( 2 + 4 \times 3 \) according to this two-tiered grammar yields:

\[ 6 \mid 8 \mid 7 \mid 6 \mid 5 \mid 4 \mid 3 \mid 2 \mid 1 \mid 0 \rightarrow D \\
\rightarrow D \mid D \times T \rightarrow T \\
\rightarrow T \mid T + E \rightarrow E \]

Parsing the string \( 2 + 4 \times 3 \) according to this grammar yields the value 14. Instead of the correct value 14.

Consider the language of parenthesis-free arithmetic expressions over digits 

\[ E \rightarrow E + D \\
\rightarrow E \times D \\
\rightarrow D \\
\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \]

A possible grammar for this language is

Capturing Operator Precedence
Easy Reading of Grammars

Consider the extended grammar $E ightarrow E + T 
| E T 
| T D 
| T = \{ \{ D * \} D = T 
| / \ . 

A term $T$ is a sequence of digits $D$ separated by the operators + or -.

An expression $E$ is a sequence of terms $T$ separated by the operators + or -.

We can interpret it as capturing the following definition:

$\begin{array}{c|c|c|c|c|c|c|c|c|c}
D & 6 & 8 & 2 & 6 & 5 & 4 & 3 & 7 & 1 \\
L & \{D * \} & D = T & / & T + & \{L + \} & L = E & - & \{L + \} & L = E
\end{array}$

Consider the extended grammar Easy Reading of Grammars
A.Pnueli

Capturing Associativity

A grammar such as

\{\{q =: p\} =: c\} =: q =: a

as

\{\{q =: p =: c =: q =: a\}

as

\{\{q =: p =: c\}

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

A := p i

A := p i

In contrast, the grammar

\{\{\{q =: p\} =: c\}

as

\{\{q =: p\}

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

6 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 ← D

z | ... | v ← p i

D =: p i | V =: p i ← V

\{\{q =: p\} =: c\}

as

\{\{q =: p\}

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

6 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 ← D

D | D + E ← E