Compilers: Principles, Techniques, and Tools

Jing-Shin Chang
Department of Computer Science & Information Engineering
National Chi-Nan University
Goals

■ What is a Compiler? Why? Applications?
■ How to Write a Compiler by Hands?
  ◆ Theories and Principles behind compiler construction - Parsing, Translation & Compiling
  ◆ Techniques for Efficient Parsing
■ How to Write a Compiler with Tools
Table of Contents

1. Introduction

2. A Simple Compiler

3. Lexical Analysis: Regular Expression and Scanner

4. Syntax Analysis: Grammars and Parsing

5. Top-Down Parsing: LL(1)

6. Bottom-Up Parsing: LR(1)
Table of Contents

7. Syntax-Directed Translation
8. Semantic Processing
9. Symbol Tables
10. Run-time Storage Organization
11. Translation of Special Structures

* . Declarations

* . Expressions and Data Structure References

* . Control Structures

* . Procedures and Functions

12. General Translation Scheme: Attribute Grammars
Table of Contents

13. Code Generation
14. Global Optimization
15. Tools: Compiler Compiler
The Compiler

■ What is a compiler?
  ◆ A program for translating programming languages into machine languages
    ✓ source language => target language

■ Why compilers?
  ◆ Filling the gaps between a programmer and the computer hardware
Compiler

Source Code (P.L., Formal Spec.)

Compiler

Target Code (P.L., Assembly, Machine Code)

Error Message
Interpreter

- Compile and then execute for each incoming statements
  - Do not save compiled coded
    - Save storage
  - Re-compile if loop back
    - Slower
  - Detect errors as one occurs during the execution time
Hybrid method & Virtual machine

- Compile into a platform independent code
- Execute the code with a virtual machine
Just-in-time Compilation

- Compile a new statement (only once) as it comes
  - And save the compiled code
  - Executed by virtual/real machine
  - Do not re-compile as it loop back
Programming Languages vs. Natural Languages

- Natural languages: for communication between native speakers of the same or different languages
  - Chinese, English, French, Japanese

- Programming languages: for communication between programmers and computers
  - Generic:
    - Basic, Fortran, COBOL, Pascal, C/C++, Java
  - Typesetting Languages:
    - TROFF (+TBL, EQN, PIC), La/Tex, PS
  - Structured Documents:
    - SGML, HTML, XML, ...
  - Script languages:
    - Csh, bsh, awk, perl, python, javascript, asp, jsp, php
Compiler: A Bridge Between PL and Hardware

<table>
<thead>
<tr>
<th>Applications (High Level Language)</th>
<th>Compiler</th>
<th>Operating System (Low Level Language)</th>
<th>Hardware</th>
</tr>
</thead>
</table>

Analysis-Synthesis Model of a Compiler

- Analysis : Program => Constituents => I.R.
  - Lexical Analysis: linear => token
  - Syntax Analysis: hierarchical, nested => tree
  - Semantic Analysis: check for legal meanings

- Synthesis: I.R. => Target Language
  - Intermediate Code Generation
  - Code Optimization:
    - machine independent + machine dependent
  - Code Generation
Typical Modules of a Compiler
position := initial + rate * 60

**lexical analyzer**

id₁ := id₂ + id₃ * 60

**syntax analyzer**

id₁ := id₂ + id₃ * 60

**semantic analyzer**

id₁ := id₂ + id₃ * 60

**code optimizer**

temp1 := id₃ * 60.0
temp1 := id₃ * 60.0

**code generator**

id₁ := id₂ + temp1

MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
Symbol Table Management

- Symbols & Symbol Table:
  - Variable name, procedure names, constant literals
  - A record for each name describing its attributes
  - Managing Information about names
    - Variable attributes:
      - Type, register/storage allocated, scope
    - Procedure names:
      - Number and types of arguments
      - Method of argument passing
        - By value, address, reference
Lexical Analysis: Tokenization

I saw the girls

I see (+ed) the girl (+s)

final := initial + rate * 60

id₁ := id₂ + id₃ * 60
Syntax Analysis: Structure

I see (+ed) the girl (+s)

NP  verb  NP
    /      /
 I see (+ed) the girl (+s)

id₁ := id₂ + id₃ * 60

Syntax Analysis

id₁ := e
id₂ + t
id₃ * 60
Semantic Analysis: Annotation

Sentence

NP  verb  NP

I see (+ed) the girl (+s)

Semantic Analysis

Subject  verb  object

I see (+ed) the girl (+s)
Semantic Checking

Sentence

Subject  Verb  Object

I see (+ed) the girl (+s)

See[+ed](I, the girl[+s])

(semantically meaningful)

Kill/Kiss (John, the Stone)

(semantically meaningless
unless the Stone refers to an animate entity)

Semantic Constraints:

◆ Agreement: (somewhat syntactic)
  ∘ Subject-Verb: I have, she has/had, I do have, she does not
  ∘ Quantifier-noun: a book, two books

◆ Selectional Constraint:
  ◆ Kill ⇔ Animate
  ◆ Kiss ⇔ Animate
Intermediate Code Generation

I see (+ed) the girl (+s)

See[+ed](I, the girl[+s])

Logic Form

Action(+anim,+anim)

Attribute evaluation

Intermediate Code Generation

temp1 := i2r ( 60 )
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
Syntax-Directed Translation (1)

- Translation from input to target can be regarded as attribute evaluation.
  - Evaluate attributes of each node, in a well defined order, based on the particular piece of sub-tree structure (syntax) wherein the attributes are to be evaluated.

- Attribute: the particular property associated with a tree node (a node may have several attributes)
  - Abstract representation of the sub-tree rooted at that node
  - The attributes of the root node represent the particular properties of the whole input statement or sentence.
    - E.g., value associated with a mathematic sub-expression
    - E.g., machine codes associated with a sub-expression
    - E.g., translation associated with a sub-sentence
Syntax-Directed Translation (2)

- Synthesis Attributes:
  - Attributes that can be evaluated based on the attributes of children node
    - E.g., value of math. expression can be acquired from the values of sub-expressions (and the operators being applied)
      - \( a := b + c \times d \) (\( \iff a.\text{val} = b.\text{val} + \text{tmp.}\text{val} \) where \( \text{tmp.}\text{val} = c.\text{val} \times d.\text{val} \))
      - \( \text{girls} = \text{girl} + s \) (\( \iff \text{tr.}\text{girls} = \text{tr.}\text{girl} + \text{tr.}\text{s} = \text{女孩+們} \iff \text{女孩們} \))

- Inherited Attributes:
  - Attributes evaluatable from parent and/or sibling nodes
    - E.g., data type of a variable can be acquired from its left-hand side type declaration or from the type of its left-hand side brother
      - \( \text{int } a, b, c; \) (\( \iff a.\text{type} = \text{INT} \& b.\text{type} = a.\text{type} \& \ldots \))
Syntax-Directed Translation (3)

- Attribute evaluation order:
  - Any order that can evaluate the attribute AFTER all its dependent attributes are evaluated will result in correct evaluation.
  - General: topological order
    - Analyze the dependency between attributes and construct an attribute tree or forest
    - Evaluate the attribute of any leave node, and mark it as “evaluated”, thus logically remove it from the attribute tree or forest
    - Repeat for any leave nodes that have not been marked, until no unmarked node
Code Optimization
[Normalization]

Was_Kill[+ed](Bill, John)
See[+ed](I, the girl[+s])

Kill[+ed](John, Bill)
See[+ed](I, the girl[+s])

Normalization into equivalent form (optional)

temp1 := i2r ( 60 )
temp2 := id₃ * temp1
temp3 := id₂ + temp2
id₁ := temp3

temp1 := id₃ * 60.0
id₁ := id₂ + temp1
Code Generation

Lexical: 看到 [了] (我, 女孩 [們])
Structural: 我 看到 女孩 [們] [了]

See[+ed](I, the girl[+s])

Selection of target words & order of phrases

Code Generation

temp1 := id_3 * 60.0
id_1 := id_2 + temp1

movf id_3, r2
mulf #60.0, r2
movf id_2, r1
addf r2, r1
movf r1, id_1

Selection of usable codes & order of codes
Objectives of Optimizing Compilers

- Correct codes
- Better performance
  - Maximum Execution Efficiency
  - Minimum Code Size
    - Embedded systems
  - Minimizing Power Consumptions
    - Mobile devices
    - Typically, faster execution also implies lower power
- Reasonable compilation time
- Manageable engineering efforts
Optimization for Computer Architectures

- Parallelism
  - Instruction level: multiple operations are executed simultaneously
  - Processor level: different threads of the same application are run on different processors

- Memory hierarchy
  - No storage that is both fast and large
Structure of a Compiler

- Front End: Source Dependent
  - Lexical Analysis
  - Syntax Analysis
  - Semantic Analysis
  - Intermediate Code Generation
  - (Code Optimization: machine independent)

- Back End: Target Dependent
  - Code Optimization
  - Target Code Generation
Structure of a Compiler

Fortran \rightarrow \text{Intermediate Code} \rightarrow \text{MIPS}

Pascal \rightarrow \text{Intermediate Code} \rightarrow \text{SPARC}

C \rightarrow \text{Intermediate Code} \rightarrow \text{Pentium}
History

- 1st Fortran compiler: 1950s
- efficient? (compared with assembly program)
  - not bad, but much easier to write programs
  - high-level languages are feasible.
- 18 man-year, ad hoc structure
- Today, we can build a simple compiler in a few month.
- Crafting an efficient and reliable compiler is still challenging.
Cousins of the Compiler

- Preprocessors: macro definition/expansion
- Interpreters
  - Compiler vs. interpreter vs. just-in-time compilation
- Assemblers: 1-pass / 2-pass
- Linkers: link source with library functions
- Loaders: load executables into memory
- Editors: editing sources (with/without syntax prediction)
- Debuggers: symbolically providing stepwise trace
- Profilers: gprof (call graph and time analysis)
- Project managers: IDE
  - Integrated Development Environment
- Deassemblers, Decompilers: low-level to high-level language
Applications of Compilation Techniques

- Virtually any kinds of Programming Languages and Specification Languages with Regular and Well-defined Grammatical Structures will need a kind of compiler (or its variant, or a part of it) to analyze and then process them.
Applications of Lexical Analysis

- Text/Pattern Processing:
  - grep: get lines with specified pattern
  - sed: stream editor, editing specified patterns
  - tr: simple translation between patterns (e.g., uppercases to lowercases)
  - AWK: pattern-action rule processing
    - pattern processing based on regular expression
Applications of Lexical Analysis

- Search Engines/Information Retrieval
  - full text search, keyword matching, fuzzy match

- Database Machine
  - fast matching over large database
  - database filter

- Fast & Multiple Matching Algorithms:
  - Optimized/specialized lexical analyzers (FSA)
  - Examples: KMP, Boyer-Moore (BM), …
Applications Syntax Analysis

- Structured Editor/Word Processor
  - Integrated Develop Environment (IDE)
    - automatic formatting, keyword insertion
  - Incremental Parser vs. Full-blown Parsing
    - incremental: patching analysis made by incremental changes, instead of re-parsing or re-compiling

- Pretty Printer: beautify nested structures
  - cb (C-beautifier)
  - indent (an even more versatile C-beautifier)
Applications Syntax Analysis

- Static Checker/Debugger: lint
  - check errors without really running
    - statement not reachable
    - used before defined
Application of Optimization Techniques

■ Data flow analysis
  ◆ Software testing:
    ◆ Locating errors before running (static checking)
    ◆ Locate errors along all possible execution paths
      • not only on test data set
  ◆ Type Checking
    ◆ Dereferncing null or freed pointers
    ◆ “Dangerous” User supplied strings
  ◆ Bound Checking
    ◆ Security vulnerability: buffer over-run attack
    ◆ Tracking values of pointers across procedures
  ◆ Memory management
    ◆ Garbage collection
Applications of Compilation Techniques

- Pre-processor: Macro definition/expansion
- Active Webpages Processing
  - Script or programming languages embedded in webpages for interactive transactions
  - Examples: JavaScript, JSP, ASP, PHP
  - Compiler Apps: expansion of embedded statements, in addition to web page parsing
- Database Query Language: SQL
Applications of Compilation Techniques

- Interpreter
  - no pre-compilation
  - executed on-the-fly
  - e.g., BASIC

- Script Languages: C-shell, Perl
  - Function: for batch processing multiple files/databases
  - mostly interpreted, some pre-compiled
  - Some interpreted and save compiled codes
Applications of Compilation Techniques

- Text Formatter
  - Troff, LaTex, Eqn, Pic, Tbl

- VLSI Design: Silicon Compiler
  - Hardware Description Languages
    - variables => control signals / data
  - Circuit Synthesis
  - Preliminary Circuit Simulation by Software
Applications of Compilation Techniques

- VLSI Design

architecture BEHAV of SHIFTER is

    signal SIG_D: STD_LOGIC;

begin

  SIG_ASSIGNMENT: process(CLK)
  begin
    if (CLK'EVENT and CLK = '1') then
      SIG_D <= D;
      Q <= SIG_D;
    end if;
  end process SIG_ASSIGNMENT;

end BEHAV;
Advanced Applications

Natural Language Processing

- advanced search engines: retrieve relevant documents
  - more than keyword matching
  - natural language query

- information extraction:
  - acquire relevant information

- text summarization:
  - get most brief & relevant paragraphs

- text mining:
  - mining information & rules from text
Advanced Applications

- Machine Translation
  - Translating a natural language into another
  - Models:
    - Direct translation
    - Transfer-Based Model
    - Inter-lingua Model
  - Transfer-Based Model:
    - Analysis-Transfer-Generation (or Synthesis) model
Tools: Automatic Generation of Lexical Analyzers and Compilers

- Lexical Analyzer Generator: LEX
  - Input: Token Pattern specification (in regular expression)
  - Output: a lexical analyzer

- Parser Generator: YACC
  - “compiler-compiler”
  - Input: Grammar Specification (in context-free grammar)
  - Output: a syntax analyzer (aka “parser”)
Tools

■ Syntax Directed Translation engines
  ◆ translations associated with nodes
  ◆ translations defined in terms of translations of children

■ Automatic code generation
  ◆ translation rules
  ◆ template matching

■ Data flow analyses
  ◆ dependency of variables & constructs
I saw a girl in the park …
Languages

- **Alphabet** - any finite set of symbols
  - \{0, 1\}: binary alphabet

- **String** - a finite sequence of symbols from an alphabet
  - 1011: a string of length 4
  - \(\varepsilon\): the empty string

- **Language** - any set of strings on an alphabet
  - \{00, 01, 10, 11\}: the set of strings of length 2
  - \(\emptyset\): the empty set
Grammars

- The sentences in a language may be defined by a set of rules called a grammar

  \[ L: \{00, 01, 10, 11\} \]

  (the set of binary digits of length 2)

  \[ G: (0|1)(0|1) \]

- Languages of different degree of regularity can be specified with grammar of different “expressive powers”

  - Chomsky Hierarchy:
    - Regular Grammar < Context-Free Grammar < Context-Sensitive Grammar < Unrestricted
Automata

- An acceptor/recognizer of a language is an automaton which determines if an input string is a sentence in the language.
- A transducer of a language is an automaton which determines if an input string is a sentence in the language, and may produce strings as output if it is in the language.
- Implementation: state transition functions (parsing table)
Transducer

- language $L_1$
- grammar $G_1$
- accept
- construct
- language $L_2$
- grammar $G_2$
- translation
- Define / Generate
- Define / Generate
Meta-languages

♦ Meta-language: *a language used to define another language*

Different *meta-languages* will be used to define the various components of a programming language so that these components can be analyzed automatically
Definition of Programming Languages

- *Lexical tokens*: regular expressions
- *Syntax*: context free grammars
- *Semantics*: attribute grammars
- *Intermediate code generation*: attribute grammars
- *Code generation*: tree grammars
Implementation of Programming Languages

♦ Regular expressions:
  finite automata, lexical analyzer

♦ Context free grammars:
  pushdown automata, parser

♦ Attribute grammars:
  attribute evaluators, type checker and intermediate code generator

♦ Tree grammars:
  finite tree automata, code generator
Appendix: Machine Translation
Machine Translation (Transfer Approach)

- Analysis is target independent, and
- Generation (Synthesis) is source independent

IR: Intermediate Representation
Example:
Miss Smith put two books on this dining table.

**Analysis**

- Morphological and Lexical Analysis
- Part-of-speech (POS) Tagging

n. Miss
n. Smith
v. put (+ed)
q. two
n. book (+s)
p. on
d. this
n. dining table.
Example:
Miss Smith put two books on this dining table.

- Syntax Analysis

```
S
  /\    
/    
NP   VP
  /  
 V  NP
 /   
/    PP
  Miss Smith put(+ed) two book(s)  on this dining table
```
Example:
Miss Smith put two books on this dining table.

■ Transfer

◆(1) Lexical Transfer

<table>
<thead>
<tr>
<th>English</th>
<th>Chinese</th>
</tr>
</thead>
<tbody>
<tr>
<td>Miss</td>
<td>小姐</td>
</tr>
<tr>
<td>Smith</td>
<td>史密斯</td>
</tr>
<tr>
<td>put (+ed)</td>
<td>放</td>
</tr>
<tr>
<td>two</td>
<td>兩</td>
</tr>
<tr>
<td>book (+s)</td>
<td>書</td>
</tr>
<tr>
<td>on</td>
<td>在...上面</td>
</tr>
<tr>
<td>this</td>
<td>這</td>
</tr>
<tr>
<td>dining table</td>
<td>餐桌</td>
</tr>
</tbody>
</table>
Example:
Miss Smith put two books on this dining table.

- Transfer

  - (2) Phrasal/Structural Transfer

小姐史密斯放兩書在上面這餐桌
史密斯小姐放兩書在這餐桌上面
Example:
Miss Smith put two books on this dining table.

- Generation: Morphological & Structural

史密斯小姐把兩本書放在這張餐桌上面
史密斯小姐放兩(本)書在這(張)餐桌上面
史密斯小姐(把)兩(本)書放在這(張)餐桌上面

中文翻譯：⇒ 史密斯小姐把兩本書放在這張餐桌上面
source program

lexical analyzer

syntax analyzer

semantic analyzer

symbol-table manager

intermediate code generator

code optimizer

code generator

target program

error handler

[Aho 86]
position := initial + rate * 60

\[
\text{lexical analyzer}
\]

id_1 := id_2 + id_3 * 60

\[
\text{syntax analyzer}
\]

\[
\text{semantic analyzer}
\]

```
\[
\text{inttoreal}
\]
```
C
intermediate code generator

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

code optimizer

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

code generator

\[
\begin{align*}
\text{Binary Code}
\end{align*}
\]
Detailed Steps (1): Analysis

- Text Pre-processing (separating texts from tags)
  - Clean up garbage patterns (usually introduced during file conversion)
  - Recover sentences and words (e.g., \texttt{<B>C</B> omputer})
  - Separate Processing-Regions from Non-Processing-Regions (e.g., File-Header-Sections, Equations, etc.)
  - Extract and mark strings that need special treatment (e.g., Topics, Keywords, etc.)
  - Identify and convert markup tags into internal tags (de-markup; however, markup tags also provide information)

- Discourse and Sentence Segmentation
  - Divide text into various primary processing units (e.g., sentences)
  - Discourse: Cue Phrases
  - Sentence: mainly classify the type of “Period” and “Carriage Return” in English (“sentence stops” vs. “abbreviations/titles”)

72
Detailed Steps (2): Analysis (Cont.)

- **Stemming**
  - English: perform morphological analysis (e.g., -ed, -ing, -s, -ly, re-, pre-, etc.) and Identify root form (e.g., got <get>, lay <lie/lay>, etc.)
  - Chinese: mainly detect suffix lexemes (e.g., 孩子們, 學生們, etc.)
  - Text normalization: Capitalization, Hyphenation, …

- **Tokenization**
  - English: mainly identify split-idiom (e.g., turn NP on) and compound
  - Chinese: Word Segmentation (e.g., [土地] [公有] [政策])
  - Regular Expression: numerical strings/expressions (e.g., twenty millions), date, … (each being associated with a specific type)

- **Tagging**
  - Assign Part-of-Speech (e.g., n, v, adj, adv, etc.)
  - Associated forms are basically independent of languages starting from this step
Detailed Steps (3): Analysis (Cont.)

- Parsing
  - Decide suitable syntactic relationship (e.g., PP-Attachment)

- Decide Word-Sense
  - Decide appropriate lexicon-sense (e.g., River-Bank, Money-Bank, etc.)

- Assign Case-Label
  - Decide suitable semantic relationship (e.g., Patient, Agent, etc.)

- Anaphora and Antecedent Resolution
  - Pronoun reference (e.g., “he” refers to “the president”)
Detailed Steps (4): Analysis (Cont.)

- Decide Discourse Structure
  - Decide suitable discourse segments relationship (e.g., Evidence, Concession, Justification, etc. [Marcu 2000].)

- Convert into Logical Form (Optional)
  - Co-reference resolution (e.g., “president” refers to “Bill Clinton”), scope resolution (e.g., negation), Temporal Resolution (e.g., today, last Friday), Spatial Resolution (e.g., here, next), etc.
  - Identify roles of Named-Entities (Person, Location, Organization), and determine IS-A (also Part-of) relationship, etc.
  - Mainly used in inference related applications (e.g., Q&A, etc.)
Detailed Steps (5): Transfer

- Decide suitable Target Discourse Structure
  - For example: Evidence, Concession, Justification, etc. [Marcu 2000].

- Decide suitable Target Lexicon Senses
  - Sense Mapping may not be one-to-one (sense resolution might be different in different languages, e.g. “snow” has more senses in Eskimo)
  - Sense-Token Mapping may not be one-to-one (lexicon representation power might be different in different languages, e.g., “DINK”, “鼹”, etc). It could be 2-1, 1-2, etc.

- Decide suitable Target Sentence Structure
  - For example: verb nominalization, constitute promotion and demotion (usually occurs when Sense-Token-Mapping is not 1-1)

- Decide appropriate Target Case
  - Case Label might change after the structure has been modified
  - (Example) verb nominalization: “… that you (AGENT) invite me” ⇔ “… your (POSS) invitation”
Detailed Steps (6): Generation

- Adopt suitable Sentence Syntactic Pattern
  - Depend on Style (which is the distributions of lexicon selection and syntactic patterns adopted)

- Adopt suitable Target Lexicon
  - Select from Synonym Set (depend on style)

- Add “de” (Chinese), comma, tense, measure (Chinese), etc.
  - Morphological generation is required for target-specific tokens

- Text Post-processing
  - Final string substitution (replace those markers of special strings)
  - Extract and export associated information (e.g., Glossary, Index, etc.)
  - Restore customer’s markup tags (re-markup) for saving typesetting work