Compiler Structure

Phases of Compilation

- Compilation process is partitioned into a series of four distinct sub-problems called **phases**, each with a separate well-defined translation task.
- This structure is the result of more than 40 years of experience in compiler engineering.
- Can be run in sequence, in which case they are **passes**.
- Or can be integrated in various ways, such as “parallel” execution as **co-routines**.
Phase 1: Lexical Analysis

Lexical Analysis

- Lexical analysis, or scanning/screening, partitions source text into the “words” or tokens of the input language.

Source Program Text

Lexical Analysis

Token Stream

if x>5 then —> keyword "if"
identifier "x"
operator ">
integer "5"
keyword "then"
Phase 2: Syntax Analysis

Syntax Analysis

- Syntax analysis, or parsing, groups tokens together into the grammatical structures of the language, often represented as a parse tree or (equivalently) a postfix token stream.

```
Token Stream

Syntax Analysis

Parse Tree or Postfix Token Stream
```

```
identifier "x"
operator ">
integer "5"

op ">"

id "x"
int "5"
```

```
identifier "x"
integer "5"
operator ">"
```
Phase 3: Semantic Analysis

Semantic Analysis

- Semantic analysis, or context analysis, analyzes and verifies the meaning of the structures, and checks for semantic legality.

Parse Tree \( \rightarrow \) Postfix Token Stream

- Semantic Analysis
- Annotated Parse Tree
- Intermediate Code

\[ \text{op } "+" \]
\[ \text{int } "2" \]
\[ \text{string } ""\text{hi}"" \]

\( \rightarrow \) ERROR
**Phase 4: Code Generation**

**Code Generation**
- Generates target machine code for analyzed structures
- May include **optimization** of structures, operations and code

```
Annotated Parse Tree or Intermediate Code

Code Generation

Target Machine or Assembly Code
```
Alternate Phase 4: Interpretation

Bytecode Interpretation

- Simulates the virtual machine of the intermediate code in software
- Executes the intermediate code directly

![Diagram of Bytecode Interpretation]

Intermediate Code

\[\text{Bytecode Interpretation}\]

Execution
Compiler Architectures

Merging Phases

• Some compilers merge or integrate one or more phases together (although they are still designed separately) to gain efficiency
• If all phases are merged, we have a one pass compiler
• One pass compilers can be very fast, but they can also be much more complex and difficult to maintain

Refining Phases

• In order to reduce complexity and increase maintainability and reliability, very high quality production compilers (e.g. IBM’s) often to the opposite - they actually split phases into even more independent sub-tasks
• For example, the IBM PL/I compiler actually has about 40 separate phases, each a separate pass
Tables

Why is it called a “Compiler”?

• Language processing involves compiling tables of information about the program, to be used to analyze scopes, look up references and determine meaning - hence the term “compiler”

• Such tables may be shared between phases of compilation, using databases or disk files

• Or in other compilers each phase builds its own tables

• Either way, a big part of compiler engineering is the design of the table structures to support efficient analysis
PT Pascal

The PT Pascal Compiler

• Originally designed and implemented by J. Alan Rosselet (U. of Toronto) about 1980, as a proof of concept for the S/SL compiler technology (then a new research result)

• Specifically designed for teaching and learning about compilers

• Designed to be easily portable (Pascal was at that time the standard portable language, as C is now) - it takes an expert about three days to port PT to a new machine (mostly spent understanding the new machine’s assembly language)

• Designed to be real - generates good quality code for target machines, can compile and run itself, the S/SL Processor, and other quite large real programs

• Self-compiling - implemented entirely in PT Pascal itself, using S/SL
PT Compiler
Structure

Four Phase, Three Pass

- Scanner / Screener run as co-routine with Parser in Parser pass
- Parse tree represented as postfix stream of tokens
- Special PT “abstract” (virtual) machine code (“T-code”)

PT Pascal Source Text

Scanner/ Screener
Parser

Postfix Token Stream

Semantic Analysis

T-code

Code Generation

x86 Assembly Code

Intel x86 Assembler

x86 Machine Code

T-code VM Simulator

Interpreted execution

(Not part of PT compiler)

(Scanner/Screener and Parser run as co-routines)

(Intermediate virtual machine code designed for PT)

( E.G. \( x + y \) \( \Rightarrow \) \( x y + \) )
PT Phase Overview: Scanner

**Scanner**

- Breaks up input text into PT language tokens, ignores blanks, newlines etc.

PT Pascal Source Text

- Input to Scanner: `a := b + 11;`
- Output from Scanner:
  - pIdentifier "a"
  - pColonEquals
  - pPlus
  - pInteger "11"
  - pSemicolon

(The “p” prefix on token names indicates that they are to be/parser input stream tokens)
PT Phase Overview: Screener

Screener

• A filter between the Scanner and the Parser
• Recognizes keywords, evaluates literals, builds identifier table

PT Pascal Source Text

Parser

Scanner/ Screener

Input to Scanner

const x=22;

Output from Scanner

pIdentifier "const"
pIdentifier "x"
pEquals
pInteger "22"
pSemicolon

Output from Scanner/Screener to Parser

Output from Scanner/ Screener to Parser

(after screening)

(const after screening)

pConst
pIdentifier 273
pEquals
pInteger 22
pSemicolon

CISC 458 Winter 2015
© 2015 J.R. Cordy
Lecture 3
**PT Phase Overview: Parser**

**Parser**
- Context-free *syntax checking* (syntax error detection)
- Converts **expressions** to expression trees in postfix form, recognizes **statement structure**, disambiguates **operators**

---

**Input to Scanner/Screener**
```
  a := b + 11;
```

**PT Pascal Source Text**
```
a := b + 11;
```

**Input to Parser from Scanner/Screener**
```
plIdentifier 632
pColonEquals
plIdentifier 243
pPlus
pInteger 11
pSemicolon
```

**Output from Parser**
```
sAssignmentStmt
sIdentifier 632
sIdentifier 243
sInteger 11
sAdd
sExpnEnd
```

(The "s" prefix to token names indicates that they are **Semantic pass input stream tokens)**
PT Phase Overview: Semantic Analyzer

Semantic Phase

- Checks semantic constraints (e.g., type checking)
- Constructs symbol table, analyzes scopes, resolves references
- Generates T-code representation of program

Postfix Token Stream

Semantic Analysis

T-code
PT Phase Overview: Semantic Analyzer

Input to Semantic Analyzer from Parser

- sAssignmentStatement
- sIdentifier 632
- sIdentifier 243
- sInteger 11
- sAdd
- sExpnEnd

Output from Semantic Analyzer

- tAssignBegin
- tLiteralAddress 120
- tLiteralAddress 150
- tFetchInteger
- tLiteralAllInteger 11
- tAddInteger
- tAssignInteger

(PT virtual machine bytecode - \textit{T}-code)
The PT Interpreter

**T-code Virtual Machine Simulator**

- The **T-code** output by the Semantic Analyzer can be executed directly by a **PT VM Simulator**, to make a PT Interpreter.

![Diagram](image)
PT Phase Overview: Code Generator

**Code Generator**

- Translates the T-code form of the program into *machine code* for the target computer
- Our target computer is the *Intel x86*, and the code generator outputs *Linux x86 assembly code*

\[\text{T-code} \rightarrow \text{Code Generation} \rightarrow \text{x86 Assembly Code}\]
PT Phase Overview: Code Generator

Input from Semantic Analyzer

- tAssignBegin
- tLiteralAddress 120
- tLiteralAddress 150
- tFetchInteger
- tLiteralAllInteger 11
- tAdd
- tAssignInteger

Output from Code Generator

- movl u+150, %eax
- addl $11, %eax
- movl %eax, u+120

(x86 assembly code)
Summary

Compiler Structure

• Compilers split into four phases:
  Lexical, Syntactic, Semantic, Code generation
• Phases may be implemented as separate passes, or integrated for performance
• PT Compiler implements four phases in three passes, using token streams between passes
• Uses PT virtual machine T-code as intermediate code

References

• Text, chapter 2

Next Time

• The Syntax/Semantic Language (S/SL) - a domain-specific language (DSL) for specifying compiler phases