Compilers

Pierre Geurts

2015-2016

E-mail	:	p.geurts@ulg.ac.be
URL	:	http://www.montefiore.ulg.ac.be/~geurts/compil.html
Bureau	:	I 141 (Montefiore)
Téléphone	:	04.366.48.15

Contact information

- Teacher: Pierre Geurts
  - p.geurts@ulg.ac.be, I141 Montefiore, 04/3664815
- Teaching assistant: Cyril Soldani
  - soldani@run.montefiore.ulg.ac.be, I.77b Montefiore, 04/3662699
- Website:
  - Course: http://www.montefiore.ulg.ac.be/~geurts/Cours/compil/2015/compil2015_2016.html
  - Project: http://www.montefiore.ulg.ac.be/~info0085

Course organization

- "Theoretical" course
  - Wednesday, 14h-16h, R18, Institut Montefiore
  - About 6-7 lectures
  - Slides online on the course web page (available before each lecture)
  - Give you the basis to achieve the project (and a little more)
- Project
  - One (big) project
  - Implementation of a compiler (from scratch) for a particular language.
  - A few repetition lectures on Wednesday, 16h-18h (checkpoints for your project).
  - (more on this later)
- Evaluation
  - Mostly on the basis of the project
  - Written report, short presentation (TBC), oral exam

References

- Books:
  - **Compilers: Principles, Techniques, and Tools (2nd edition), Aho, Lam, Sethi, Ullman, Prentice Hall, 2006**
    `http://dragonbook.stanford.edu/`
  - Modern compiler implementation in Java/C/ML, Andrew W. Appel, Cambridge University Press, 1998
    `http://www.cs.princeton.edu/~appel/modern/`
  - Engineering a compiler (2nd edition), Cooper and Torczon, Morgan Kaufmann, 2012.
- On the Web:
  - **Basics of compiler design, Torben Aegidius Mogensen, Self-published, 2010**
    `http://www.diku.dk/hjemmesider/ansatte/torbenm/Basics/index.html`
  - Compilation - Théorie des langages, Sophie Gire, Université de Brest
    `http://www.lisyc.univ-brest.fr/pages_perso/leparc/Etud/Master/Compil/Doc/CoursCompilation.pdf`
  - Standford compilers course
    `http://www.stanford.edu/class/cs143/`

Course outline

Part 1:	Introduction
Part 2:	Lexical analysis
Part 3:	Syntax analysis
Part 4:	Semantic analysis
Part 5:	Intermediate code generation
Part 6:	Code generation
Part 7:	Conclusion

Part 1

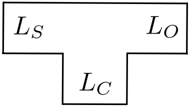
Introduction

Outline

1. What is a compiler
2. Compiler structure
3. Course project

Compilers

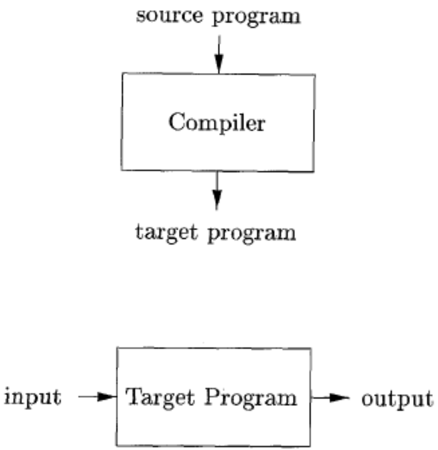
- A compiler is a program (written in a language L_c) that:
  - reads another program written in a given source language L_s
  - and translates (compiles) it into an equivalent program written in a second (target) language L_O.



- The compiler also returns all errors contained in the source program

- Examples of combination:
  - L_C=C, L_S =C, L_O=Assembly (gcc)
  - L_C=C, L_S =java, L_O=C
  - L_C=java, L_S =LaTeX, L_O=HTML
  - ...
- Bootstrapping: $L_C = L_S$

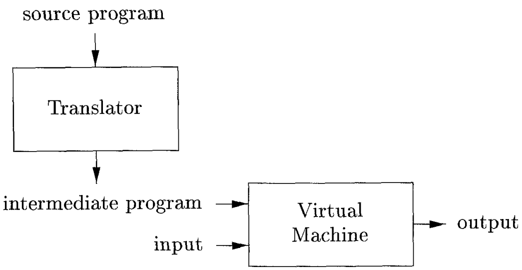
Compiler


source program
Compiler
target program
input
Target Program
output

Interpreter

- An interpreter is a program that:
  - executes directly the operations specified by the source program on input data provided by the user
- Usually slower at mapping inputs to outputs than compiled code (but gives better error diagnostics)

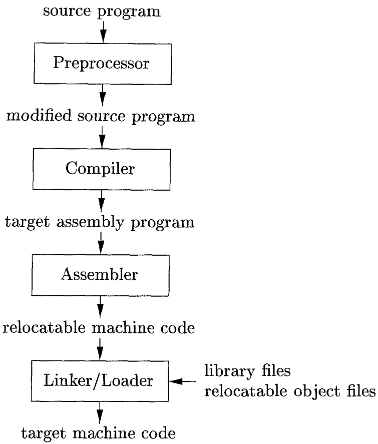
Hybrid solution



- Hybrid solutions are possible
- Example: Java combines compilation and interpretation
    - Java source program is compiled into an intermediate form called *bytecodes*
    - Bytecodes are then interpreted by a java virtual machine (or compiled into machine language by *just-in-time* compilers).
- Main advantage is portability

A broader picture

- Preprocessor: include files, macros... (small compiler).
- Assembler: generate machine code from assembly program (small trivial compiler).
- Linker: relocates relative addresses and resolves external references.
- Loader: loads the executable file in memory for execution.



Why study compilers?

- There is small chance that you will ever write a full compiler in your professional carrier.
- Then why study compilers?
  - To improve your culture in computer science (not a very good reason)
  - To get a better intuition about high-level languages and therefore become a better coder
  - Compilation is not restricted to the translation of computer programs into assembly code
    - Translation between two high-level languages (Java to C++, Lisp to C, Python to C, etc.)
    - Translation between two arbitrary languages, not necessarily programming ones (word to html, pdf to ps, etc.), aka source-to-source compilers or transcompilers

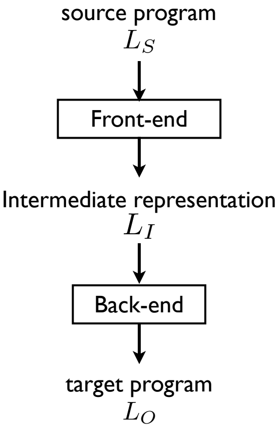
Why study compilers?

- The techniques behind compilers are useful for other purposes as well
  - Data structures, graph algorithms, parsing techniques, language theory...
- There is a good chance that a computer scientist will need to write a compiler or an interpreter for a domain-specific language
  - Example: database query languages, text-formatting language, scene description language for ray-tracers, search engine, sed/awk, substitution in parameterized code...
- Very nice application of concepts learned in other courses
  - Data structures and algorithms, introduction to the theory of computation, computation structures...

General structure of a compiler

- Except in very rare cases, translation can not be done word by word
- Compilers are (now) very structured programs
- Typical structure of a compiler in two stages:
  - Front-end/analysis:
    - Breaks the source program into constituent pieces
    - Detect syntaxic and semantic errors
    - Produce an intermediate representation of the language
    - Store in a symbol table information about procedures and variables of the source program
  - Back-end/synthesis:
    - Construct the target program from the intermediate representation and the symbol table
  - Typically, the front end is independent of the target language, while the back end is independent of the source language
  - One can have a middle part that optimizes the intermediate representation (and is thus independent of both the source and target languages)

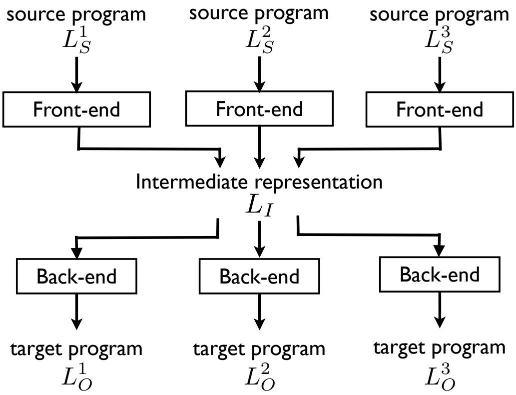
General structure of a compiler



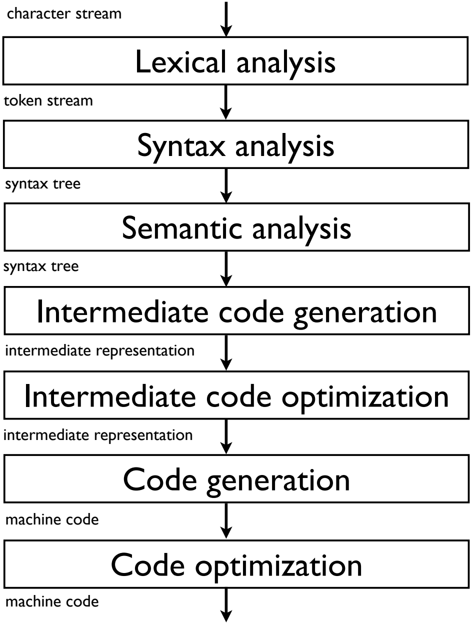
Intermediate representation

The intermediate representation:

- Ensures portability (it's easy to change the source or the target language by adapting the front-end or back-end).
- Should be at the same time easy to produce from the source language and easy to translate into the target language



Detailed structure of a compiler



Lexical analysis or scanning

Input: Character stream ⇒ **Output:** token streams

- The lexical analyzer groups the characters into meaningful sequences called lexemes.
  - Example: "`position = initial + rate * 60;`" is broken into the lexemes `position`, `=`, `initial`, `+`, `rate`, `*`, `60`, and `;`.
  - (Non-significant blanks and comments are removed during scanning)

- For each lexeme, the lexical analyzer produces as output a token of the form: $\langle \textit{token-name}, \textit{attribute-value} \rangle$
  - The produced tokens for "`position = initial + rate * 60`" are as follows

$$\langle \mathbf{id}, 1\rangle, \langle \mathbf{op}, =\rangle, \langle \mathbf{id}, 2\rangle, \langle \mathbf{op}, +\rangle, \langle \mathbf{id}, 3\rangle, \langle \mathbf{op}, *\rangle, \langle \mathbf{num}, 60\rangle$$

with the symbol table:

1	position	...
2	initial	...
3	rate	...

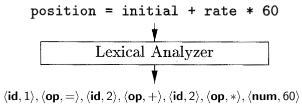
(In modern compilers, the table is not built anymore during lexical analysis)

Lexical analysis or scanning

In practice:

- Each token is defined by a regular expression
  - Example:
    $Letter = A - Z|a - z$
    $Digit = 0 - 9$
    $Identifier = Letter(Letter|Digit)^*$
- Lexical analysis is implemented by
  - building a non deterministic finite automaton from all token regular expressions
  - eliminating non determinism
  - Simplifying it
- There exist automatic tools to do that
  - Examples: lex, flex...

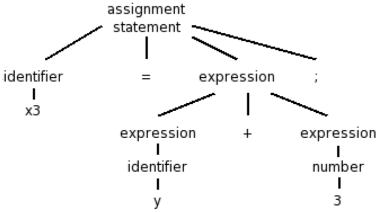
Lexical analysis or scanning



Syntax analysis or parsing

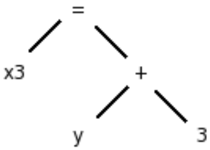
Input: token stream ⇒ **Output:** syntax tree

- Parsing groups tokens into grammatical phrases
- The result is represented in a parse tree, ie. a tree-like representation of the grammatical structure of the token stream.
- Example:
  - Grammar for assignement statement:
    asst-stmt → id = exp ;
    exp → number | id | expr + expr
  - Example parse tree:



Syntax analysis or parsing

- The parse tree is often simplified into a (abstract) syntax tree:



- This tree is used as a base structure for all subsequent phases

- On parsing algorithms:
  - Languages are defined by context-free grammars
  - Parse and syntax trees are constructed by building automatically a (kind of) pushdown automaton from the grammar
  - Typically, these algorithms only work for a (large) subclass of context-free grammars

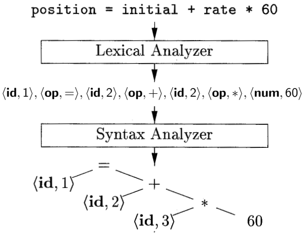
Lexical versus syntax analysis

- The division between scanning and parsing is somewhat arbitrary.
- Regular expressions could be represented by context-free grammars
- Mathematical expression grammar:

	EXPRESSION	$\rightarrow$	EXPRESSION OP2 EXPRESSION
Syntax	EXPRESSION	$\rightarrow$	NUMBER
	EXPRESSION	$\rightarrow$	(EXPRESSION)
	OP2	$\rightarrow$	$+ \mid - \mid * \mid /$
Lexical	NUMBER	$\rightarrow$	DIGIT | DIGIT NUMBER
	DIGIT	$\rightarrow$	$0\mid1\mid2\mid3\mid4\mid5\mid6\mid7\mid8\mid9$

- The main goal of lexical analysis is to simplify the syntax analysis (and the syntax tree).

Syntax analysis or parsing



Semantic analysis

Input: syntax tree ⇒ **Output:** (augmented) syntax tree

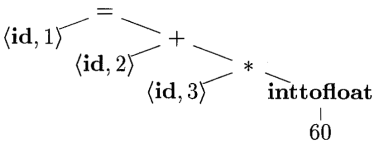
- Context-free grammar can not represent all language constraints, e.g. non local/context-dependent relations.

- Semantic/contextual analysis checks the source program for semantic consistency with the language definition.
  - A variable can not be used without having been defined
  - The same variable can not be defined twice
  - The number of arguments of a function should match its definition
  - One can not multiply a number and a string
  - ...

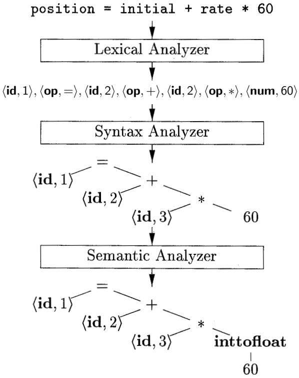
  (none of these constraints can be represented in a context-free grammar)

Semantic analysis

- Semantic analysis also carries out type checking:
  - Each operator should have matching operands
  - In some cases, type conversions (coercions) might be possible (e.g., for numbers)
- Example: `position = initial + rate * 60`
  If the variables `position`, `initial`, and `rate` are defined as floating-point variables and 60 was read as an integer, it may be converted into a floating-point number.



Semantic analysis



Intermediate code generation

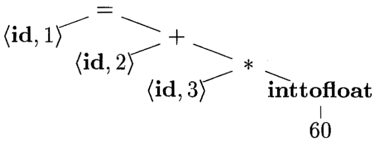
Input: syntax tree $\Rightarrow$ **Output:** Intermediate representation

- A compiler typically uses one or more intermediate representations
  - Syntax trees are a form of intermediate representation used for syntax and semantic analysis
- After syntax and semantic analysis, many compilers generate a low-level or machine-like intermediate representation
- Two important properties of this intermediate representation:
  - Easy to produce
  - Easy to translate into the target machine code

Intermediate code generation

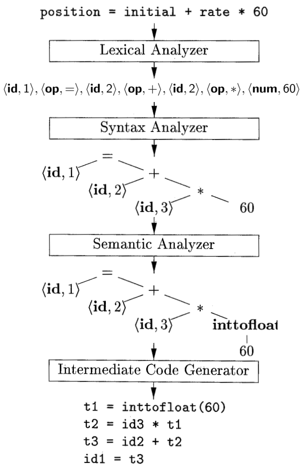
- Example: Three-address code with instructions of the form `x = y op z`.
  - Assembly-like instructions with three operands (at most) per instruction
  - Assumes an unlimited number of registers

- Translation of the syntax tree




```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Intermediate code generation



Intermediate code optimization

Input: Intermediate representation ⇒ **Output:** (better) intermediate representation

- Goal: improve the intermediate code (to get better target code at the end)
- Machine-independent optimization (versus machine-dependent optimization of the final code)
- Different criteria: efficiency, code simplicity, power consumption…
- Example:

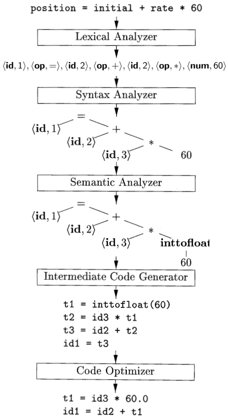
```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

⇒

```
t1 = id3 * 60.0
id1 = id2 + t1
```

- Optimization is complex and very time consuming
- Very important step in modern compilers

Intermediate code optimization



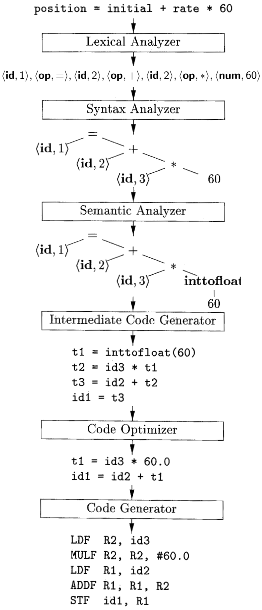
Code generation

Input: Intermediate representation $\Rightarrow$ **Output:** target machine code

- From the intermediate code to real assembly code for the target machine
- Needs to take into account specifities of the target machine, eg., number of registers, operators in instruction, memory management.
- One crucial aspect is register allocation
- For our example:

```
t1 = id3 * 60.0
id1 = id2 + t1
⇒
LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1,R1
```

Final code generation



Symbol table

1	position	...
2	initial	...
3	rate	...

- Records all variable names used in the source program
- Collects information about each symbol:
  - Type information
  - Storage location (of the variable in the compiled program)
  - Scope
  - For function symbol: number and types of arguments and the type returned
- Built during lexical analysis (old way) or in a separate phase (modern way).
- Needs to allow quick retrieval and storage of a symbol and its attached information in the table
- Implementation by a dictionary structure (binary search tree, hash-table,...).

Error handling

- Each phase may produce errors.
- A good compiler should report them and provide as much information as possible to the user.
  - Not only "syntax error".
- Ideally, the compiler should not stop after the first error but should continue and detect several errors at once (to ease debugging).

Phases and Passes

- The description of the different phases makes them look sequential
- In practice, one can combine several phases into one pass (i.e., one complete reading of an input file or traversal of the intermediate structures).
- For example:
  - One pass through the initial code for lexical analysis, syntax analysis, semantic analysis, and intermediate code generation (front-end).
  - One or several passes through the intermediate representation for code optimization (optional)
  - One pass through the intermediate representation for the machine code generation (back-end)

Compiler-construction tools

- First compilers were written from scratch, and considered as very difficult programs to write.
  - The first fortran compiler (IBM, 1957) required 18 man-years of work

- There exist now several theoretical tools and softwares to automate several phases of the compiler.
  - Lexical analysis: regular expressions and finite state automata (Software: (f)lex)
  - Syntax analysis: grammars and pushdown automata (Softwares: bison/yacc, ANTLR)
  - Semantic analysis and intermediate code generation: syntax directed translation
  - Code optimization: data flow analysis

This course

- Although the back-end is more and more important in modern compilers, we will insist more on the front-end and general principles
- Outline:
  - Lexical analysis
  - Syntax analysis
  - Semantic analysis
  - Intermediate code generation (syntax directed translation)
  - Some notions about code generation and optimization

Compiler project

(Subject to changes)

- Implement a “complete” compiler
- By group of 1, **2**, or 3 students
- Source language: Cool, The **C**lassroom **O**bject-**O**riented **L**anguage
  - An academic object-oriented programming language, to which you will be asked to add features
  - `http://theory.stanford.edu/~aiken/software/cool/cool.html`
- The destination language will be LLVM, a popular modern intermediate language
  - `http://llvm.org/`
- Implementation language L_c can be chosen among c, c++, java, python, javascript, ocaml, scheme, and lisp.