Formal Languages and Compilers
Lecture I: Introduction to Compilers

Alessandro Artale

Free University of Bozen-Bolzano
Faculty of Computer Science – POS Building, Room: 2.03
artale@inf.unibz.it
http://www.inf.unibz.it/∼artale/

Formal Languages and Compilers — BSc course

2018/19 – Second Semester

Course Overview

- Introduction to the Notion of Compiler.
- Formal Language Theory: Chomsky Classification and notion of Formal Grammar.
- Theory of regular languages: deterministic and non-deterministic finite automata, Regular expressions and Regular grammars.
- Context-free languages and their grammars.
- Lexical Analysis and Automata.
- Syntax Analysis and Parsers:
  - Top-Down Parser
  - Bottom-Up Parser
- Syntax-Directed Translation to Translate Programming Language Constructs.
- Semantic Analysis: Type Checking.
- Code Generation and Principles of Code Optimization.

Final Exam

- Final Written Exam: 70% of the total mark
- Mid-Term Exam: Grants the possibility to skip the Formal Language part of the final exam.
- Compiler Project: 30% of the total mark
  - Form teams of two/tree persons
  - Decide and implement your little language developing a compiler for it.
  - Two weeks after the end of the course you will present a demo of your project.
  - You are free to develop your project either in C or Java.

Reading List

Introduction to Automata Theory, Languages, and Computation (3rd edition), J.E. Hopcroft, R. Motwani, J.D. Ullman. Addison Wesley, 2007.
Compilers: Principles, Techniques, and Tools, Alfred V. Aho, Ravi Sethi and Jeff Ullman. Publisher: Prentice Hall, 2003.

Further reading material:
Compiler Construction: Principles and Practice, Kenneth C. Louden. Publisher: Brooks Cole, 1997.
Programming Language Processors in Java: Compilers and Interpreters, David Watt and Deryck Brown. Publisher: Prentice Hall, 2000.
Advanced Compiler Design and Implementation, Steven Muchnick. Publisher: Morgan Kaufmann, 1997.

Summary of Lecture I

- Motivations and Brief History.
- The Architecture of a Compiler.
- The Analysis Phase.
- The Synthesis Phase.
- Towards Executable Code: Assembler, Loader and Linker.

How are Languages Implemented?

- Two major strategies:
  1. **Compilers.** Translate programs to a machine executable code. They do extensive preprocessing.
  2. **Interpreters.** Run programs "as is" without preliminary translation: Successive phases of translation (to machine/intermediate code) and execution.

History of High-Level Languages

- 1953 IBM develops the 701: All programming done in assembly.
  - Problem: Software costs exceeded hardware costs!
- John Backus: *Speedcoding*: An interpreted language that ran 10-20 times slower than hand-written assembly!
- John Backus: Translate high-level code to assembly
  - Many thought this impossible. Had already failed in other projects.
  - 1954-7 FORTRAN I project: By 1958, $> 50\%$ of all software is in FORTRAN. Cut the development time dramatically (from weeks to hours).

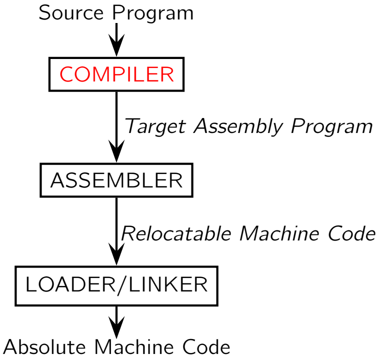
Summary

- Motivations and Brief History.
- The Architecture of a Compiler.
- The Analysis Phase.
- The Synthesis Phase.
- Towards Executable Code: Assembler, Loader and Linker.

The Context of a Compiler

A *compiler* is a program that reads a program written in one language–the *source* language–and translates it into an equivalent program in another language–the *target* language.

In addition to a compiler, other programs are needed to generate an *executable code*.



The Architecture of a Compiler

Compilation can be divided in two parts: Analysis and Synthesis.

1. **Analysis.** Breaks the source program into constituent pieces and creates intermediate representation.
2. **Synthesis.** Generates the target program from the intermediate representation.

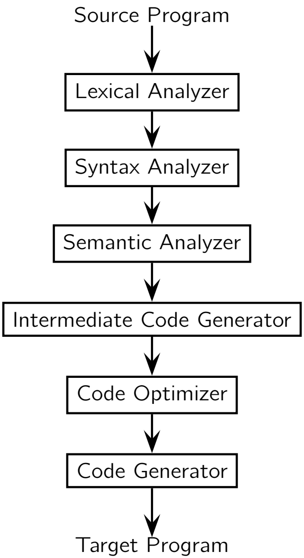
The analysis part can be divided along the following phases:

1. **Lexical Analysis**;
2. **Syntax Analysis**;
3. **Semantic Analysis**.

The synthesis part can be divided along the following phases:

1. **Intermediate Code Generator**;
2. **Code Optimizer**;
3. **Code Generator**.

The Architecture of a Compiler (Cont.)



Summary

- Motivations and Brief History.
- The Architecture of a Compiler.
- The Analysis Phase.
- The Synthesis Phase.
- Towards Executable Code: Assembler, Loader and Linker.

Lexical Analysis

- The program is considered as a unique sequence of characters.
- The Lexical Analyzer reads the program from left-to-right and sequence of characters are grouped into tokens–lexical units with a collective meaning.
- The sequence of characters that gives rise to a token is called lexeme.

Lexical Analysis: An Example

Let us consider the following assignment statement:

$$\texttt{position} = \texttt{initial} + \texttt{rate} * 60$$

Then, the lexical analyzer will group the characters in the following tokens:

Lexeme	Token
`position`	ID
=	=
`initial`	ID
+	+
`rate`	ID
*	*
60	NUM

Symbol Table

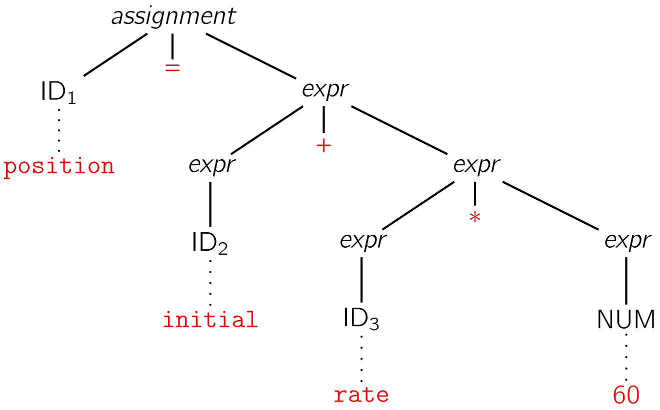
- An essential function of a compiler is to build the Symbol Table where the identifiers used in the program are recorded along with various properties:
    - Storage allocated for the ID; its type; its scope (where in the program is valid); number and types of its arguments (in case the ID is a procedure name); etc.
- When an identifier is detected an ID token is generated, the corresponding lexeme is entered in the Symbol Table, and a pointer to the position in the Symbol Table is associated to the ID token.

Syntactic Analysis

- The Syntactic Analysis is also called Parsing.
- Tokens are grouped into grammatical phrases represented by a Parse Tree which gives a hierarchical structure to the source program.
- The hierarchical structure is expressed by recursive rules, called Grammar's Productions.
- **Example.** Grammar's Productions for assignment statements are:

$$
\begin{aligned}
<assignment> &\rightarrow \text{ID } ``=" <expr> \\
<expr> &\rightarrow \text{ID} \mid \text{NUM} \mid <expr><op><expr> \mid (<expr>) \\
<op> &\rightarrow + \mid - \mid * \mid /
\end{aligned}
$$

Parse Tree: An Example



Grammars and Formal Language Theory

- The notion of Grammar is related to studies in natural languages.
- Linguists were concerned with:
  1. Defining the valid sentences of a Language;
  2. Providing a structural definition of such valid sentences.
- The Formal Language Theory considers a Language as a mathematical object.
- A Language is just a set of strings. To formally define a Language we need to formally define what are the strings admitted by the Language:
  - A Grammar is a formalism that gives a finite representation of a Language and allows to generate the set of strings belonging to a given Language.

Semantic Analysis

- The Semantic Analysis phase checks the program for semantic errors (Type Checking) and gathers type information for the successive phases.
- **Type Checking.** Check types of operands (possibly imposing type coercions); No real number as index for array; etc.

Summary

- Motivations and Brief History.
- The Architecture of a Compiler.
- The Analysis Phase.
- The Synthesis Phase.
- Towards Executable Code: Assembler, Loader and Linker.

Intermediate Code Generation

- An intermediate code is generated as a program for an abstract machine.
- The intermediate code should be easy to translate into the target program.
- As intermediate code we consider the *three-address code*, similar to assembly: sequence of instructions with at most *three* operands such that:
  1. There is at most one operator, in addition to the assignment. Thus, we make explicit the operators precedence.
  2. Temporary names must be generated to compute intermediate operations.

Example. The intermediate code for the assignment statement is:

$$\mathtt{temp1} = \mathtt{inttoreal}(60)$$

$$\mathtt{temp2} = \mathtt{id3} * \mathtt{temp1}$$

$$\mathtt{temp3} = \mathtt{id2} + \mathtt{temp2}$$

$$\mathtt{id1} = \mathtt{temp3}$$

Code Optimization

- This phase attempts to improve the intermediate code so that faster-running machine code can be obtained.
- Different compilers adopt different optimization techniques.

Example. A simple optimization of the intermediate code for the assignment statement could be:

$$\begin{array}{lcl} \mathtt{temp1} = \mathtt{inttoreal}(60) & & \\ \mathtt{temp2} = \mathtt{id3} * \mathtt{temp1} & & \mathtt{temp1} = \mathtt{id3} * 60.0 \\ & \text{---}> & \\ \mathtt{temp3} = \mathtt{id2} + \mathtt{temp2} & & \mathtt{id1} = \mathtt{id2} + \mathtt{temp1} \\ \mathtt{id1} = \mathtt{temp3} & & \end{array}$$

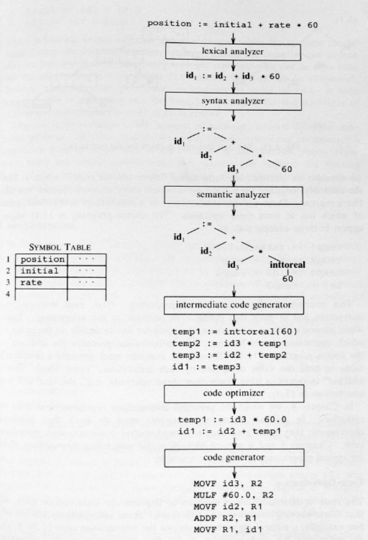
Code Generation

- This phase generates the target code consisting of assembly code.
  1. Memory locations are selected for each variable;
  2. Instructions are translated into a sequence of assembly instructions;
  3. Variables and intermediate results are assigned to memory registers.

Example. A target code generated from the optimized code of the assignment statement could be:

`MOVF id3, R2`	The `F` stands for floating-point instruction
`MULF #60.0, R2`	The # means that 60.0 is a constant
`MOVF id2, R1`	The first and second operand of each instruction
`ADDF R2, R1`	specify a source and a destination
`MOVF R1, id1`	

Summing Up


position := initial + rate * 60
lexical analyzer
id_1 := id_2 + id_3 * 60
syntax analyzer
:=
id_1
+
id_2
*
id_3
60
semantic analyzer
:=
id_1
+
id_2
*
id_3
inttoreal
60
SYMBOL TABLE
1 position ...
2 initial ...
3 rate ...
4
intermediate code generator
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
code optimizer
temp1 := id3 * 60.0
id1 := id2 + temp1
code generator
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1

Summary

- Motivations and Brief History.
- The Architecture of a Compiler.
- The Analysis Phase.
- The Synthesis Phase.
- Towards Executable Code: Assembler, Loader and Linker.

Assembler

- The Assembler is responsible for translating the target code–usually assembly code–into an executable machine code.
- The assembly code is a mnemonic version of machine code in which:
  1. Names are used instead of binary codes for operations (*Code Table*).
  2. Names are used for operands instead of memory locations (*Symbol Tables*).

Loader and Linker

- The machine code generated by the Assembler can be executed only if allocated in Main Memory starting from the address “0”.
- Since this is not possible the Loader will alter the relocatable addresses of the code to place both instructions and data in the right place in Main Memory.
- The starting free address, `L`, in Main Memory to allocate the program is called the *Relocation Factor*. The Loader must:
    1. Add to each relocatable address the relocation factor `L`;
    2. Leave unaltered each absolute address–e.g., address of I/O devices.
- The Linker links together the different files/modules of a single program and, possibly, adds library files.

Summary of Lecture I

- Motivations and Brief History.
- The Architecture of a Compiler.
- The Analysis Phase.
- The Synthesis Phase.
- Towards Executable Code: Assembler, Loader and Linker.