Compiler Theory: Introduction

Marc Moreno Maza
University of Western Ontario
CS447b – Winter 2004

December 1, 2004

1 Course Outline

1.1 Course Description

The emphasis of this course is on solving problems universally encountered in designing a language translator.

Few people are likely to build or even maintain a compiler. However, the ideas and techniques of compiler theory apply to many problems in software engineering and in other areas. For example, the string matching techniques for building the lexical analyzer of a compiler are also used by text editors or pattern matching programs. The techniques from graph theory for performing code optimization, such as coloring, are used to organize schedules.

Hence, the student should come away from the course with a deeper understanding of what happens when they compile programs but also with an enriching experience with algorithms and computers.

1.2 Course Contents

- Lexical analysis
- Parsing
- Syntax directed translations

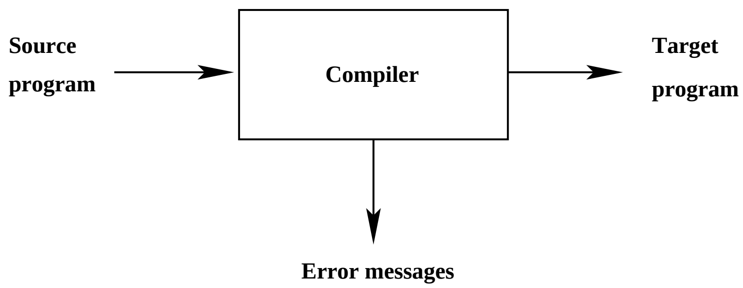

Figure 1: A compiler.

- Type checking
- Run-time organization
- Intermediate code generation
- Code generation

2 Compilers

2.1 Definition

Examples: `cc`, `latex`, `latex2html`, `a2ps`.

2.2 The Analysis-Synthesis Model

- The ANALYSIS part (Figure 2)
  - breaks up the source program into constituent pieces (words, phrases)
  - and creates an *intermediate representation* of the source program.

  Informally, the compiler must understand the structure and meaning of the source program.

- The INTERMEDIATE REPRESENTATION (IR) is an abstract machine language
  - that can express the target-machine operations,
  - without committing too much machine-specific detail.

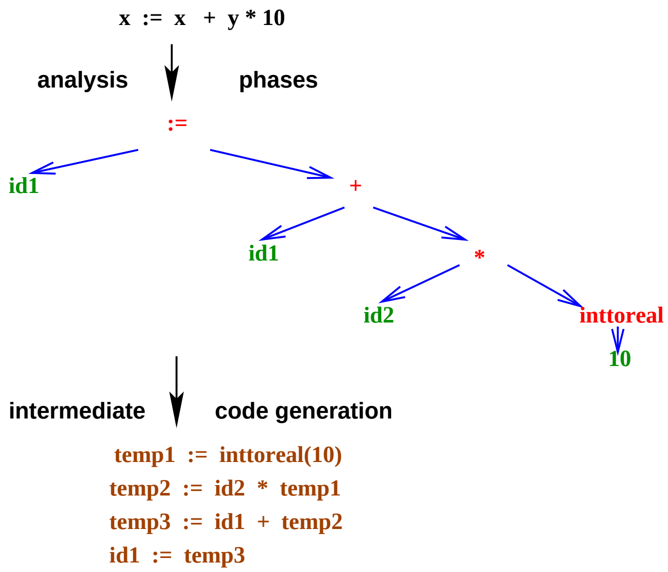

Figure 2: From a source line to an intermediate representation via an expression tree. In this example it is assumed that the variables `x` and `y` are known to be of type `real` which explains the conversion of `10` from `int` to `real`.

- The IR should be easy
  - to produce (from the source program) and
  - to translate (to the target language).
- In our example (Figure 2) the IR is a <u>THREE-ADDRESS CODE</u>:
  - each instruction has at most 3 operands,
  - each instruction has at most 1 operator in addition to the assignment.

  On the right picture of Figure 3 the arrows going from programming languages to IR are called <u>COMPILER FRONT ENDS</u> and those going from IR to computer architectures are called <u>COMPILER BACK ENDS</u>.
- The <u>SYNTHESIS</u> part (Figure 4)
  - performs code optimization,

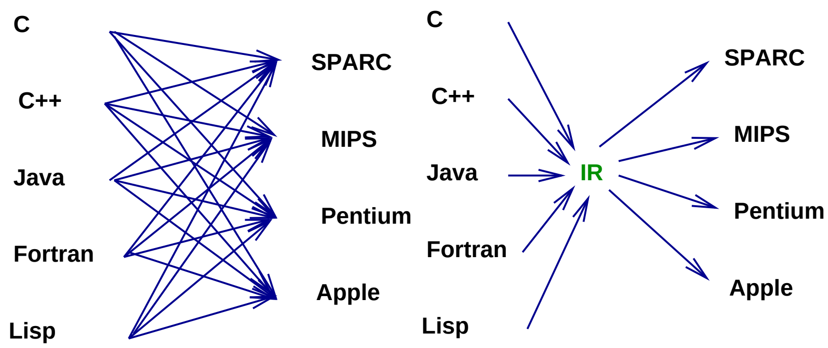

Figure 3: The interest of an intermediate representation.

- and constructs the desired target program from the intermediate representation.

2.2.1 The analysis phases

The analysis consists of three steps (Figure 5).

- The LINEAR ANALYSIS in which the stream of characters making up the source program
  - is read from left-to-right
  - and converted into a stream of words, where a word is a sequence of characters with a collective meaning.
- The HIERARCHICAL ANALYSIS in which words
  - are grouped into nested collections (= grammatical phrases) with a collective meaning
  - represented by a PARSE TREE
- The SEMANTICAL ANALYSIS in which certain checks are performed
  - to ensure that the components of a program fit together meaningfully (type checking, type conversion, ...)
  - and to report on errors.

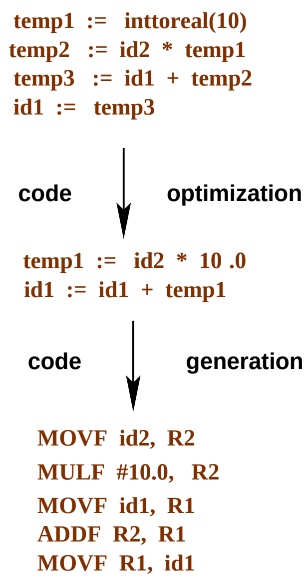

Figure 4: From an intermediate representation to a sequence of target instructions.

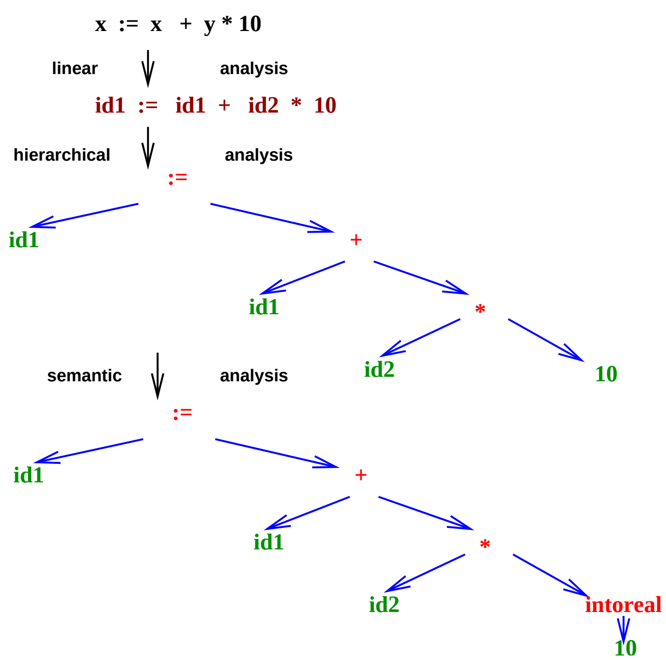

Figure 5: The analysis phases.

2.2.2 Tokens, patterns and lexemes

- The words generated by the linear analysis may be of different kinds:
  - identifier,
  - keyword (`if`, `while`, ...),
  - punctuation character,
  - multi-character operator (`:=`, `->`, ...).
- Such a kind is called a TOKEN and an element of a kind is called a LEXEME.
- A word is recognized to be a lexeme for a certain token by PATTERN MATCHING. For instance *letter followed by letters and digits* is a pattern that matches a word like `x` or `y` with the token *id* (= identifier).

Token	**Lexeme**	**Pattern**
ID	x y n0	letter followed by letters and digits
NUM	-123 1.456e-5	any numeric constant
IF	if	if
LPAREN	(	(
LITERAL	“Hello”	any string of characters (except “) between “ and “

2.2.3 The synthesis phases

- The CODE OPTIMIZATION attempts to improve the intermediate code in order to reduce the running time and the running space of the target executable program.
- Some optimizations are trivial. In our example
  - the conversion of the integer `10` to the float `10.0` can be avoided,
  - the creation of the TEMPORARY VARIABLES `temp1` and `temp3` can be avoided too.
- Other optimizations require more sophisticated techniques ...
- The CODE GENERATION consists of

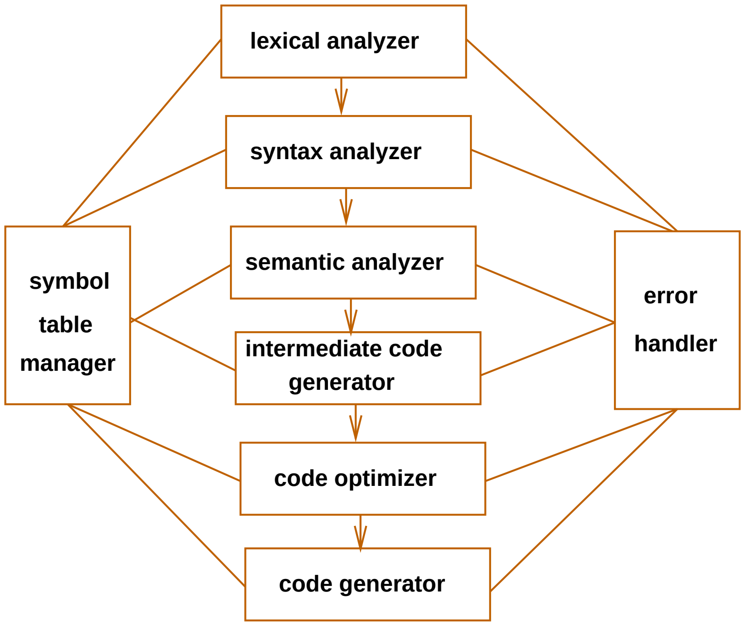

Figure 6: The structure of a compiler.

- selecting memory location for each variable used by the (optimized) program
- translating the IR into a sequence of target instructions.

- In our example
  - registers `R1` and `R2` are chosen,
  - the first and the second operand of each instruction specify a source and a destination, respectively.
  - the `F` of each instruction specifies that the instruction deals with floats
  - ...

2.3 Structure of a compiler

Any large software is easier to understand and implement if it is divided into well-defined modules.

- In a compiler,

  - linear analysis
    * is called LEXICAL ANALYSIS or SCANNING and
    * is performed by the LEXICAL ANALYZER or LEXER,
  - hierarchical analysis
    * is called SYNTAX ANALYSIS or PARSING and
    * is performed by the SYNTAX ANALYZER or PARSER.

- During the analysis, the compiler manages a SYMBOL TABLE by
  - recording the identifiers of the source program
  - collecting information (called ATTRIBUTES) about them: storage allocation, type, scope, and (for functions) signature.

- When the identifier `x` is found by the lexical analyzer
  - generates the token *id*
  - enters the lexeme `x` in the symbol-table (if it is not already there)
  - associates to the generated token a pointer to the symbol-table entry `x`. This pointer is called the LEXICAL VALUE of the token.

- During the analysis or synthesis, the compiler may DETECT ERRORS and report on them.
  - However, after detecting an error, the compilation should proceed allowing further errors to be detected.
  - The syntax and semantic phases usually handle a large fraction of the errors detectable by the compiler.

2.4 Additional components: preprocessors, assemblers, loaders/link-editors

Several other programs may be required to produce an EXECUTABLE target program.

- A PREPROCESSOR produces an input to a compiler. To do so it may perform the following tasks.

Collecting files. A source program may be divided into modules stored in different files. The PREPROCESSOR will collect these modules (for instance inclusion of header files in C) and generate the source program file.

Macro processing. The preprocessor will expand shorthands, called MACROS.

Language extension. Some language facilities (data structures, case statements) may be added to the language by BUILT-IN macros expanded by the he preprocessor.

- An ASSEMBLER translates
  - ASSEMBLY CODE that is a mnemonic version of machine code where names are used instead of binary codes for
    - operations,
    - memory addresses
  - into RELOCATABLE MACHINE CODE that is a machine code that can be loaded from any location ℓ of the memory (hence, if ℓ is added to all addresses in the code, all references will remain correct).
- A LOADER/LINK-EDITOR performs the following tasks.
  - The link-editor makes a single file from several files of relocatable machine code.
  - The loader
    - translates a relocatable machine code into an absolute machine code,
    - places it in memory at the proper locations.

We consider now an example. The above file `addAndPrint.c` is a C program. We compile it with the `cc` compiler using the `-save-temps` option which allows to save temporary files. In our example these files are

- `addAndPrint.i` which is the output of the preprocessor,
- `addAndPrint.s` which is the assembly code generated by cc,

- `addAndPrint.o` which is the relocatable machine code generated by the assembler of cc from `addAndPrint.s`,
- `a.out` which is the executable machine code generated by the linker from `addAndPrint.o`.

```
[moreno@iguanodon tmp]$ more addAndPrint.c

#define ADD +
#define PRINT__floats__x__and__y printf("x = % d y = % d\n", x, y);

main()
{
  int x,y;
  x=20; y=3;
  x = x ADD y*10;
  PRINT__floats__x__and__y;
}

[moreno@iguanodon tmp]$ cc -save-temps addAndPrint.c
[moreno@iguanodon tmp]$ ls
addAndPrint.c  addAndPrint.i  addAndPrint.o  addAndPrint.s  a.out
[moreno@iguanodon tmp]$ more addAndPrint.i

# 4 "addAndPrint.c"
main()
{
  int x,y;
  x=20; y=3;
  x = x + y*10;
  printf("x = % d y = % d\n", x, y);;
}

[moreno@iguanodon tmp]$ file addAndPrint.s
addAndPrint.s: ASCII assembler program text
[moreno@iguanodon tmp]$ more addAndPrint.s

.file "addAndPrint.c"
```

```
.version "01.01"
gcc2_compiled.:
.section .rodata
.LC0:
.string "x = % d y = % d\n"
.text
.align 4
.globl main
.type  main,@function
main:
pushl %ebp
movl %esp, %ebp
subl $8, %esp
movl $20, -4(%ebp)
movl $3, -8(%ebp)
movl -8(%ebp), %edx
movl %edx, %eax
sall $2, %eax
addl %edx, %eax
leal (%eax,%eax), %edx
leal -4(%ebp), %eax
addl %edx, (%eax)
subl $4, %esp
pushl -8(%ebp)
pushl -4(%ebp)
pushl $.LC0
call printf
addl $16, %esp
leave
ret
.Lfe1:
.size  main,.Lfe1-main
.ident "GCC: (GNU) 2.96 20000731 (Red Hat Linux 7.3 2.96-110)"

[moreno@iguanodon tmp]$ file addAndPrint.o
addAndPrint.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped
[moreno@iguanodon tmp]$ file a.out
a.out: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses shared libs), not stripped
```