

Programming Language Implementation

Bernd Meyer
bernd.meyer@acm.org

Consultation: 2pm-4pm, Tuesdays, Rm 115, Bldg 26

Basics of Programming Language Implementation.

- **Wk 5: Introduction, Lexical Analysis.**
- **Wk 6: Grammars, Top-down Parsing.**
- **Wk 7: Syntax-directed Translation.** (A 3 out)
- **Wk 8/9: Programming Paradigms.**
- **Wk 10: Bottom-up and LR(k) Parsing.** (A 3 due)
- **Wk 11: Parser Generators, Error Recovery.** (A 4 out)
- **Wk 12: Semantic Analysis and Code Generation.**
- **Wk 13: Revision Lectures for Exam.** (A 4 due)

The material is (loosely) based on Aho/Sethi/Ullman and Wilhelm and Maurer.

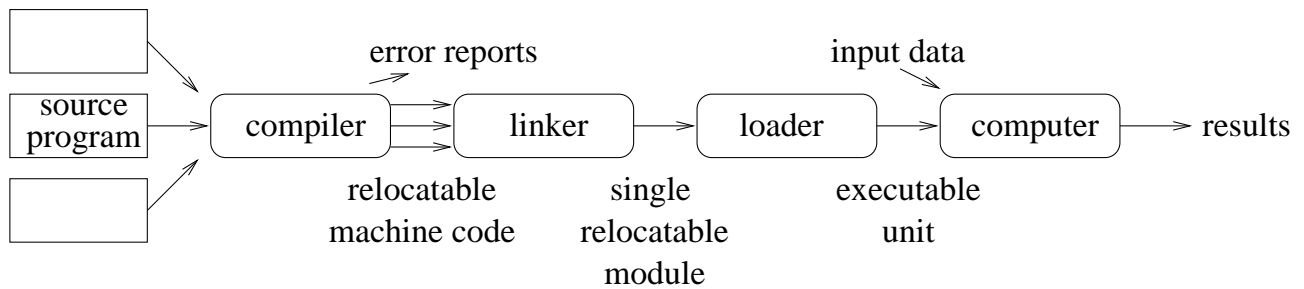
References

- Well-established field with lots of good standard material. We use
 - A.V. Aho, R. Sethi, J.D. Ullman
Compilers—Principles, Techniques and Tools. Addison Wesley, 1986.
 - A.W. Appel
Modern Compiler Implementation in ML. Cambridge University Press, 1997.
 - A. Willhelm and D. Maurer
Compiler Design. Addison Wesley, 1995.
- For the underlying theory (formal languages and automata)
J.E. Hopcroft and J.D. Ullman
Introduction to Automata Theory, Languages and Computation. Addison Wesley, 1979.
- A useful source for compiler implementations (to obtain hands-on knowledge)
<http://www.idiom.com/free-compilers/>

Compiler Construction

A compiler implements a programming language on some target machine by translating it into a language that this machine “understands” directly.

- native machine code
- virtual machine code
- other programming language

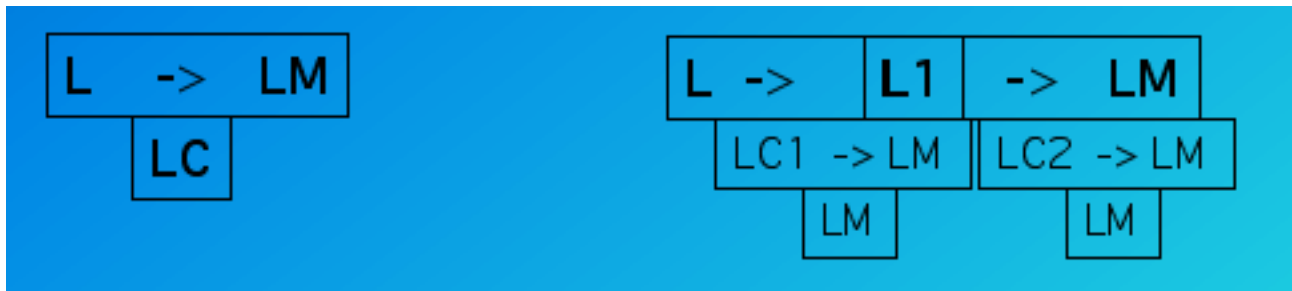


- one of the best understood areas of computer science
- combines technical application with solid formal theory
- many industrial strength (de-facto standard) tools support the process

Compiler vs. Interpreter

A compiler translates a program P_L written in language L into a program P_{LM} written in language LM . It is itself implemented in LC such that

$$\forall \text{input data } D : P_{LM}(D) = P_L(D)$$



An Interpreter I executes a program P_L written in language L together with its input data D and is itself implemented in language LM such that

$$\forall \text{input data } D : I_{LM}(P_L, D) = P_L(D)$$

	Compiler	Interpreter
Efficiency	+	
Static Checking	+	
Dynamic Code Change		+
Prototyping		+
Interactive Debugging		+

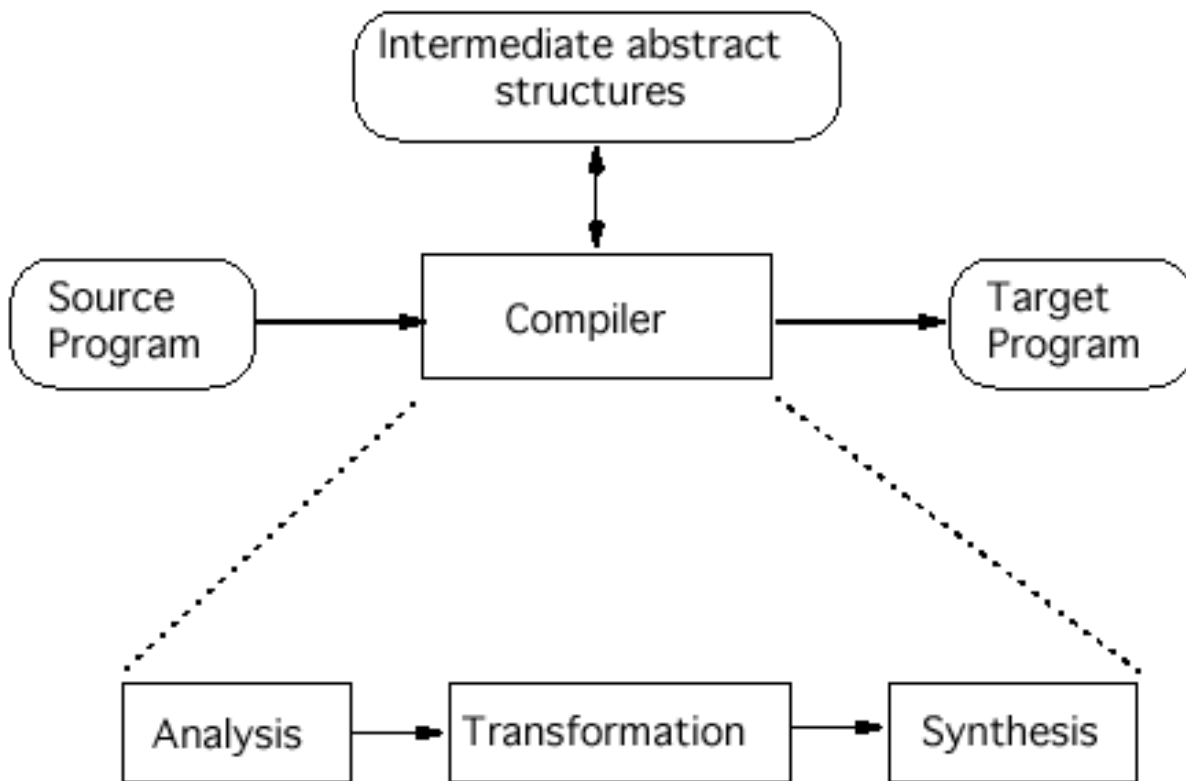
Translators

Many techniques used in compiler construction are general translator techniques and have a much wider applicability.

- Preprocessors, Macro processors
- Text formatters
- HTML2TeX etc.
- Query Interpreters
- Silicon Compilers
- Postscript converters etc.
- TeX / LaTeX
- XML reader
- XSLT
- ...

All of these include translator components, some of these can be understood as programming languages (eg. Postscript, TeX)

Structure of Compilation

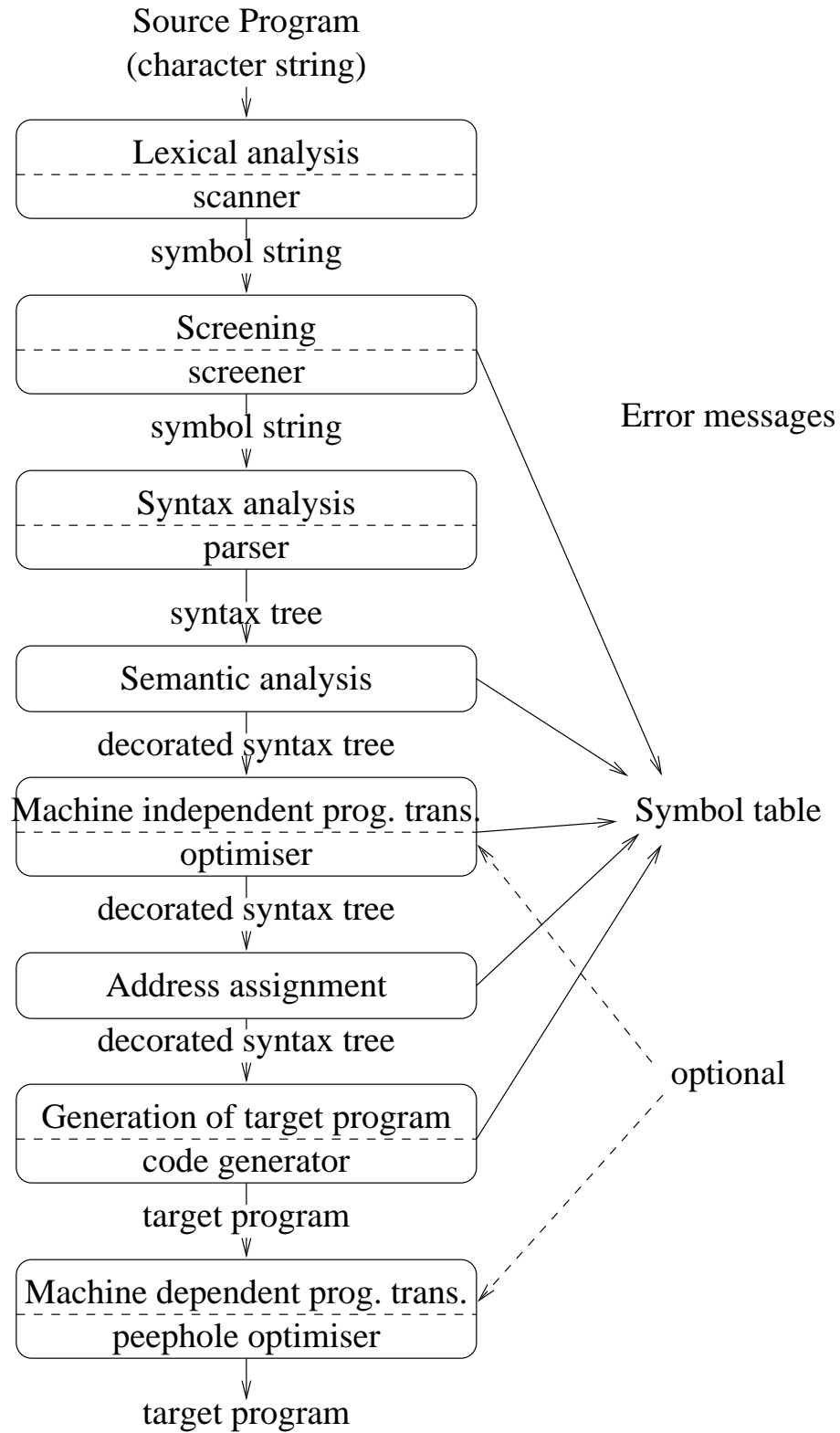


In reality the execution of these modules is not necessarily strictly sequenced.

Single-pass compilers perform all of these tasks interlinked with a single sweep over the source. This is only possible in special simple cases.

Normal compilers first construct an intermediate data structure (parse tree, see below) and traverse this for subsequent tasks.

Phases of a Compiler



Symbol Table

The central data structures are

- the **symbol table** which contains
 - identifiers
from screening
 - tags for variables
from semantic analysis
 - types for identifiers
from semantic analysis
 - addresses for variables
from address allocation
- the **parse tree**, which reflects the hierarchical program structure.

Lexical Analysis

The **scanner** (lexical analyser) converts the source program (string of characters) into a sequence of tokens (i.e. lexical units).

Tokens are, for example

- identifiers (“this”, “x”, ...)
- numbers and other constants (“3.14”, “99”, “A String t”, ...)
- operators (“+”, “-”, “(”, ...)

At the same time meaningless whitespace (blanks, tabs, line breaks, etc.) are stripped.

In some cases the scanner already strips comments, too.

Lexical Analysis

Example

```
strcpy(s, t)
  char *s, *t;
  { while(*s++ = *t++); }
```

```
[ strcpy, (, s, ,, t, ), char, *, s, ,, *, t, ;, {,
while, (, *, s, +, +, =, *, t, +, +, ),;, } ]
```

Methods

For the specification of the token-level we use regular expressions or regular (type 3) grammars

Recognition of these can be implemented with finite state automata (FSA).

Screening

Recall:

The Screener performs further analysis and classification of the tokens

This includes

- recognition of reserved words (“while”, “return” ...)
- stripping comments
- recognizing compiler directives (so-called pragmas)

Finally it creates the *symbol table* and links the tokens to the entries in the symbol table.

Scanner and Screener are in practice usually combined into a single module.

Modes of operation

- can run in a single pass before the syntax analysis
- can run interleaved with syntax analysis (on-demand)

Screening

Example

```
[ strcpy, (, s, ,, t, ), char,  
*, s, ,, *, t, ;, {,  
while, (, *, s, +, +, =,  
*, t, +, +, ),;, } ]
```

```
[ res strcpy, (, id(1), ,, id(2), ),  
res char, op(*), id(1), ,, op(*), id(2), ;,  
{, res while, (, *, ... ]
```

Methods

As for the lexical analysis we use regular grammars (type 3 grammars) or regular expressions which are implemented with finite state automata (FSA).

Additional table look-up and calls to arbitrary (programmed) functions are used to achieve the additional capabilities, such as recognition of reserved words.

Syntax Analysis

The task of the syntax analyser (called a parser) is to recover the hierarchical syntactical structure of the program which reflects its logical structure

It generates a syntax tree which is (together with the symbol table) the central data structure for all following phases.

Parsing is theoretically very well understood. The theory is based on context-free grammars (type 2 grammars) and thus pushdown automata, but it makes use of several refined classes of grammars.

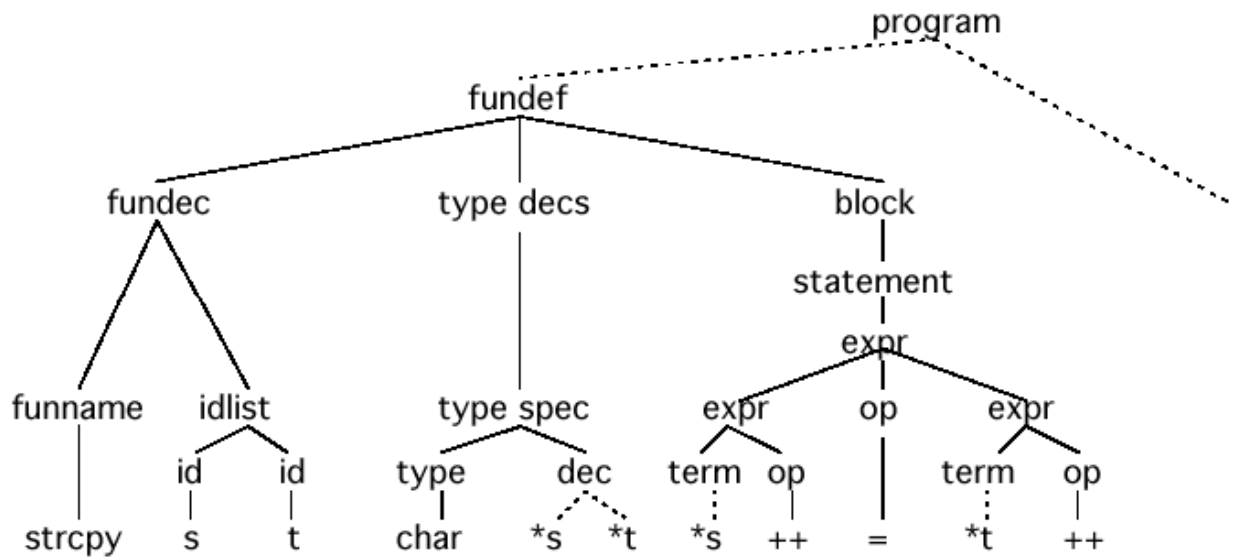
Error Handling

An important task of the syntax analyser is to check the syntactical correctness of a program and to locate, report (and diagnose) syntax errors.

Correct detection of (and recovery from) syntactical errors is one of the more difficult tasks in syntax analysis.

Parse Tree

The parse tree (syntax tree) reflects the hierarchical structure of the program.



Semantic Analysis

Determines those non-syntactic properties that can be determined from the program text.

Typically determines the **kind** of each identifier.

Which identifiers are variables?

Performs **type checking** and **type inference**.

Adds the kind and type information to the symbol table.

Machine-Independent Optimization

This uses **static analysis** to do more **checking** and to perform **efficiency increasing transformations** on the level of the source program code.

One example of **checking** is to determine that on every possible execution path each variable is initialized before being used.

Typical **code optimizations** include:

- recognizing and Eliminating tail recursion
- recognizing and eliminating constant expressions
- combining expressions and eliminating intermediate variable assignments
- avoiding re-calculation of invariant parts in loops
- converting procedure/function/method calls to in-line code

This phase is **optional**.

Address Assignment

This performs **storage allocation** and **address assignment**.

This requires information about the target machine such as the

- word length
- address length
- types of access instructions
- alignment conditions.

Variable addresses are placed in the symbol table.

Code Generation

Code generation produces the actual target program (machine instructions or VM instructions).

At this point the program must be error-free (tested by previous phases)

The central issues are

- instruction selection
eg. how is $a := a+1$ implemented (via addition ADD or via increment INC?)
- register allocation
which variables will be assigned to processor registers at which point of the program to which register will each of these be assigned?
Not all registers can perform all functions E.g. special registers used for multiplication. Results often returned in fixed register
- instruction sequencing

Example

Suppose that the target machine has instructions:

LOAD **A, R** Load contents of location with address **A** into register **R**

STORE **A, R** Store contents of register **R** into location with address **A**

LOADI **I, R** Load integer constant **I** into register **R**

MULT **A, R** Multiply the contents of location with address **A** by the contents of register **R** and store the result in **R**

And that the machine has two registers, *R1* and *R2*.

Exercise: Give target code for the example program.

Machine-dependent Optimization

In contrast to machine-independent optimization which looks at more global properties machine-dependent optimization tries to exploit properties of the target architecture to make the code more efficient.

This is usually done by improving local segments of the code with

Peephole” optimization.

The name refers to a small “peephole” passing over the code and revealing a local portion to be optimized in isolation.

Typical functions are

- changing instructions/instruction sequences to more efficient ones (special cases)
- removing unused instruction.

Abstract Machines / Virtual Machines

Instead of compiling directly into machine code, the target is often a virtual machine.

A virtual machine is essentially an interpreter for a virtual language that runs on the target machine.

The virtual language provides an intermediate level between high-level language and native machine code that is specifically designed for a particular class of languages. (e.g. procedural, object-oriented, logic, functional) and provides the basic data structures and basic control mechanisms in these languages.

As a consequence, translation into virtual machine code is easier. Native machine code is more complex, requires explicit data structure and address management, provides only very simple forms of control structures and requires more optimization.

The main arguments for using a virtual machine (such as the JVM) are

- higher portability
- security (easier encapsulation)

However, native code is typically much faster.

Summary

We have looked at the main phases in a compiler

- the function of compilers and interpreters
translation vs. execution
- the basic structure and processing phases of a compiler
 - lexical analysis
 - screening
 - parsing
 - semantic analysis
 - adress assignment
 - code generation
 - optimization
- the central data structures used in a compiler
symbol tables and parse trees

Homework

- Read Chapter 6 of Wilhelm and Maurer.
- Revise regular expressions. In particular you should revise
 - regular expressions
 - finite state automata (FSA)
 - conversion of a non-deterministic FSA into a deterministic FSA.