Compiler Construction: Introduction

Diego Linares

A compiler translates a **source language** program (high-level) to a **target language** that is usually object code. They use the same techniques as command interpreters. There is significant interaction between the structure of a compiler and the design of the programming language being compiled.

I. Why compilers? A brief history

Initially programs were written in **machine language**, which later got replaced by **assembly** which gave symbolic forms, which was translated by an **assembler**. Was more efficient to write, but still had a lot of limitations (having to be written differently for each machine).
FORTRAN was the first to have a more mathematical notation. Chomsky at the time developed **hierarchy** for the languages, based on the power of algorithms to recognize them. **Type 2 - Context free grammars** proved to be the most useful for programming languages. These are closely related to finite automata and regular expressions.
A problem that continues to this day is **code improvement techniques** to generate object code.
The study of the parsing problem allowed to create **parser generators** and the study of automata **scanner generator**.
Current advances include inferring and simplifying the info in a program. Also, being added to IDEs.

II. Programs related to compilers

A. *Interpreters*

Translate, but execute the source program immediately, which may be preferred depending on language and situation. Considerably slower.

B. *Assemblers*

Translator for assembly language of a particular computer. A compiler may translate into assembly, and let the assembler do the final part.

C. *Linkers*

Collects various object codes into a file that is directly executable. As well as connecting the source code to library function and OS calls.

D. *Loaders*

Code that has memory reference relative to an undertemined starting location are said to be **relocatable**, and the loader gives them an address. Makes the code more flexible. Usually done in conjunction with linking.

E. *Preprocessors*

Called before the actual translation, deletes comments, macro sustitutions, etc.

F. *Editors*

Oriented towards the format or structure of the language (structure-based). May inform the coder of errors before compiling.

G. *Debuggers*

Determines execution errors in a compiled program. Might also halt execution through breakpoints. These tasks are accomplished by being supplied symbolic info. by the compiler.

H. *Profilers*

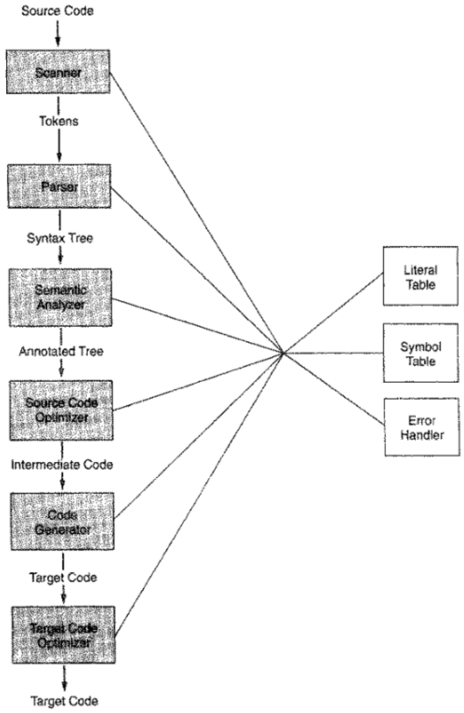
Collects statistics of the behavior of the object program during execution. May be used by the compiler to improve the object code.

I. *Project managers*

Coordinates merging of files, mantain history of changes. It can mantain info on the specific compiler and linker operations to build the program.

III. The translation process

Separated in phases that in practice may be coded together.



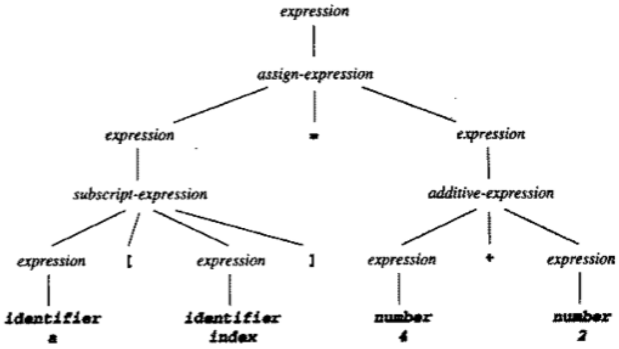
It uses 3 auxiliary components: Literal and symbol tables, and the error handler.

A. Scanner

Does **lexical analysis** by collecting characters in the form of **tokens**. May enter identifiers into the symbol table, and literals into the literal table (numeric constants).

B. Parser

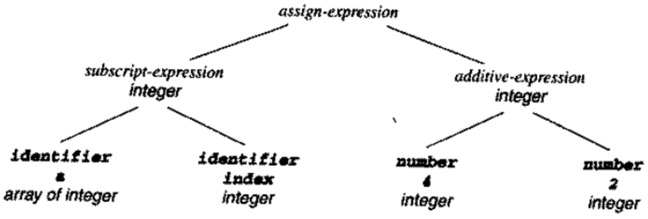
Syntax analysis determines de structure of the program, through a **parse/syntax tree**.



Internal nodes are structures, and leafs are tokens. They can be inneficient and a much simpler alternative is an **abstract syntax tree**.

C. Semantic analyzer

Determines the runtime behavior. Features determined previous to execution are **static semantics** (the dynamics can't be determined by the compiler).
Declarations and type checking generate **attributes** which are added into the tree as annotations.



D. Source code optimizer

Can be done as early as after the syntax analysis. For example, doing the sum beforehand (*constant folding*). This helps collapse the tree.
Another way of improving is the *three-adress* code.

```
// From this:
t = 4 + 2
a[index] = t
// To this
a[index] = 6
```

The syntaax tree can be referred to as a form of **intermediate code/IR**.

E. Code generator

Generates code for the target machine (mostly object code). The properties of the target machine and the representation of data are the main focus.

F. Target code optimizer

Improves target code (choosing addressing modes, replace instructions, etc.). s

IV. Major data structures in a compiler

Ideally the time to do all the previous process is $O(n)$, which is done with aid of structures.

A. Tokens

A value of an enumerated data type. Sometimes might be needed to store info. associated with it. In most languages it is only necessary to store one token at a time (**single symbol lookahead**), in some others, you might need an array.

B. Syntax tree

Standard pointer-baased structure dynamically allocated, stored as a pointer to the root. Each node's fields are the info collected by parser and semantic analyzer. Since it might have different requirements depending on the type of data, each node might be represented by a variant record.

C. Symbol table

Keeps info associated with identifiers (functions, variables, constants), which may be used to make appropriate object code choices. Since it is accesed often, a hash table or different trees are used.

D. Literal table

Stores constants and strings and allows their reuse. Does not allow deletions. Used by the code generator to contruct symbolic addresses for literals and for entering data definitions.

E. Intermediate code

May be kept in different forms depending on the optimizations done. Must pay attention to representations that allow easy reorganization.

F. Temporary files

To hold the program during compilation. Also used for **backpatch** addresses. for example, when translating a conditional. A jump from the test to the else part must be generaated before the location for the else parart is known. A blank must be left, which is done by the use of these files.

V. Other issues in compiler structure

A. Analysis and synthesis

Analysis being compute the properties of the source program (lexical and syntax analysis), and producing (generating) translated code is synthesis. Optimization includes both parts. Analysis is more mathematical while synthesis requires more specialized techniques.

B. Front end and back end

Front depends only in the source language, and back in the target one. Similar division to the previous, but in this case, target dependent optimizations are back end.
Important for **portability** since the compiler can be focused on either end. Still, most compilers tend to have parts of both. But a consistent effort to separate parts results in portability.
Front end transforms source code into *intermediate code* which the back end part uses to generate target code.

C. Passes

Each pass is a process of the entire source code, and multiples are done to generate code (to create intermediate code, then IR, etc.). Some compilers are single pass, which are faster but less efficient target code. Most range from 3 to 8.

D. Language definition and compilers

The semantics of a laanguage are usually in **language definition** written in natural language. The techniques available for the compiler highly influence these, so they are written at the same time.
In some cases the definition is so well known (C), that a programmer must interpret it and generate a compiler for it with the help of a **test suite**.
Some cases the formal definition is in maathematical terms (denotational semantics), and there should be an existent proof that a compiler works for it. Almost never done though.
The language definition also defines the complexity of the runtime environment (for example FORTRAN77 can use a static one due to lack of pointers or recursiveness).

E. Compiler options and interfaces

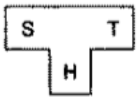
Interface such as input and output and access to the file system. And options for code optimization for example. Known as **pragmatics**, some language definitions require some to be provided (standard library in C).

F. Error handling

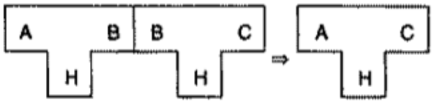
Static (at the time of compilation) errors are reported meaningfully. Each phase handles it differently, so it requires different operations.
The definition also requires some errors be caught during execution. So the compiler must generate extra code for runtime tests. More specific **exception handling** can be quite complicated.

G. Bootstraping and porting

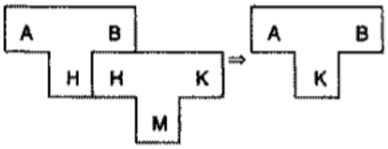
The **host language** in which the compiler is written would have to be machine. Can also be done in a language for which a compiler already exists. If the compiler runs on the target machine, we compile the new compiler using the old one and get a running program.
If the old compiler runs in a different machine than the target one. Compilation produces a **cross-compiler** (generates code for a different machine).



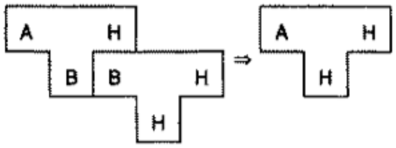
Where S is source, H is host and T is target, we expect H and T to be the same but in cases it is not, we can combine them in 2 ways.



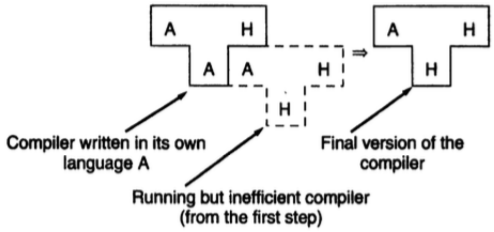
Put it through another compiler that transforms it into the one we need.



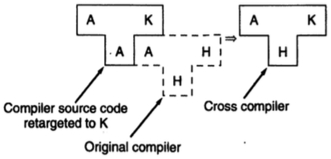
Or translate the implementation language from the host to the target one.

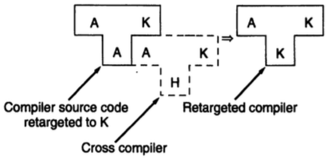


It's common to write a compiler in the language to be compiled. This might generate a circularity problem to be tackled.
We can write a quick and dirty compiler that makes correct but inefficient code, and use it to compile the good compiler. This process is **bootstrapping**. Any improvements can be added to the good compiler with the process shown before.


A
H
A
A
H
H
⇒
A
H
H
Compiler written in its own language A
Final version of the compiler
Running but inefficient compiler (from the first step)

Also porting only requires the baack end of the source code be rewritten. The old compilerr produces a cross compiler, and the compiler is recompiled by the cross oneto produce a working version to the new machine.



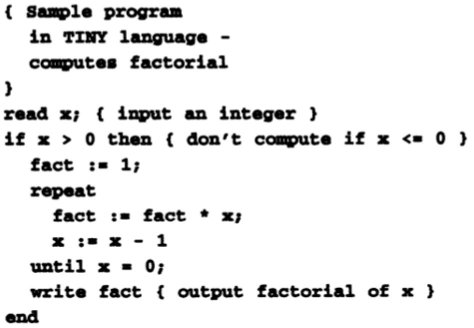


VI. THE TINY SMAPLE LANGUAGE AND COMPILER

Very simple language which compiler contains all the features any ”real” compiler needs.

A. *TINY language*

Sequence of statements separated with semicolons, variables declared by assigning values, control statements are `if repeat`.
Expressions are only boolean (but no variables of this type) or arithmetic.



```
{ Sample program
  in TINY language -
  computes factorial
}
read x; { input an integer }
if x > 0 then { don't compute if x <= 0 }
  fact := 1;
  repeat
    fact := fact * x;
    x := x - 1
  until x = 0;
  write fact { output factorial of x }
end
```

B. *TINY compiler*

It’s done in C. It’s 4 pass:

1) Scanner + parser construct the syntax tree.
2) Semantic analysis (symbol table)
3) Semantic analysis (type checking)
4) Code generator

Also has flags for `NO_PARSE, NO_ANALYZE, NO_CODE`. And executes `.tny` files. Also allows for options in information listing. `EchoSource, TestScan`, etc.

C. *TM Machine*

It’s assembly language is the target language for the compiler. Has some properties of *Reduced Instruction Set Computers*.
There’s 3 tipes of loading: `LDC, LD, LDA` (constant, memory, adress).
Arithmetic instructions can only have register operands and are ”three-address”.
The compiler still computes absolute adresses for jumps. Also the machine contains built-in I/O facilities for integers. It executes `.tm` files that are the result of running the compilator with a TINY file.

VII. C-MINUS: A LANGUAGE FOR A COMPILER PROJECT

More elaborate than TINY, restricted version of C. Contains ints, arrays of ints, functions, static declarations and simple recursive functions.