Compilers

Todd Davies

February 28, 2016

Overview

Any program written in any programming language must be translated before it can be executed on a certain piece of hardware. This translation is typically accomplished by a software system called compiler. This module aims to introduce students to the principles and techniques used to perform this translation and the key issues that arise in the construction of modern compilers.

Syllabus

Introduction What is a compiler? A high-level view of compilation. General structure of a compiler. An overview of compilation technology.

Lexical Analysis (Scanning) Regular languages/expressions, finite state machines, building regular expressions from a finite automaton.

Syntax Analysis (Parsing) Expressing Syntax, Context Free Grammars, Top-Down Parsing, Bottom-Up parsing.

Semantic Analysis Context-sensitive analysis, Attribute Grammars, Symbol Tables, Type Checking.

Intermediate Representations Properties, taxonomy, Graphical IRs, Linear IRs.

Storage Management The Procedure Abstraction, Linkage convention, Run-time storage organisation.

Code Generation Code Shape, Instruction Selection, Register Allocation, Instruction Scheduling.

Topics in Compiler Construction Code Optimisation, JIT Compilation.

Attribution

These notes are based off of both the course notes (`http://studentnet.cs.manchester.ac.uk/ugt/COMP36512/`). Thanks to Rizos Sakellariou for such a good course! If you find any errors, then I'd love to hear about them!

Contribution

Pull requests are very welcome: `https://github.com/Todd-Davies/third-year-notes`

Contents

1 Intro

A compiler is a program that reads another program written in one language and translates it into an equivalent program written in another language.

An interpreter reads the source code of a program and produces the results of executing the source. Many issues related to compilers are also present in the construction and execution of interpreters.

A good compiler exhibits the following qualities:

- It generates correct code
- It generates code that runs fast
- It confirms to the specification of the input language
- It copes with any input size, number of variables, line length etc
- The time taken to compile source code is linear in its size
- It has good diagnostics
- It has *consistent* optimisation's
- It works will with a debugger

An example of a compiler that optimises code is if we had a for-loop such as:

```
A = []
for i = 0 to N:
  A[i] = i
endfor
```

If N was always equal to 1, then the compiler should optimise this to:

```
A = [1]
```

There are two things that any sensible compiler *must* do:

- Preserve the meaning of the program being compiled
- Improve the source code in some way

In addition, it could do some (it's pretty much impossible to do all) of these things:

- Make the output code run fast
- Make the size of the output code small
- Provide good feedback to the programmer; error messages, warnings etc
- Provide good debugging information (this is hard since transforming the program from one language into another often obscures the relationship between an instance of a program at runtime and the source code it was derived from).
- Compile the code quickly

2 Parts of the compiler

In this section, we'll give a brief overview of all the bits inside a compiler. Most compilers can be roughly split into two parts; the front end and the back end:

The front end is concerned with analysis of the source language, making sure that the input is legal and producing an intermediate representation.

The back end translates the intermediate representation into the target code. This usually involves choosing appropriate instructions for each operation in the intermediate representation.

Usually, the front end can run in linear time, while the back end is `NP-Complete`. We cam automate much of the front end of the compiler.

Having an intermediate representation (IR) of the program is so helpful since it means we can completely separate the front and back ends of a compiler. Consequently, we can have multiple front ends all compiling different languages into the same IR, and multiple back ends producing instructions for different architectures.

If we have n languages and m architectures, then we'd need to write $n \times m$ monolithic compilers, but can instead write n front ends and m back ends $(n + m)$. This is *almost* as good as it sounds; in practice, you need to choose your IR very carefully, since its hard to make it encapsulate everything all the features of a language (a good example of this is compiling Scala onto code that runs on the JVM, where type erasure is a big limitation for Scala).

In more detail, we code goes through the following steps inside the compiler:

Lexical Analysis (front end):
Here, the source language is read in and tokenized (grouped into tokens for later). If the input was `a = b + x`, you might get `(id,a)(=)(id,b)(+)(id,c)` out of the other end. Whitespace is usually ignored (or, depending on the language, Incorporated into the parsing), and a symbol table is generated, containing the words that are not reserved words in the language (e.g. variable names).

Syntax Analysis (front end):
Here, the tokens are given a hierarchical structure, often using recursive rules such as Context Free Grammars. The output might be an **Abstract Syntax Tree** (AST), which is an abstract representation of the program (basically an IR).

Semantic Analysis (front end):
Here, we check for semantic errors (such as type checking, flow control checks etc). The AST is annotated with the results of the checks which can be used for optimisation later.

Intermediate code generation (front end):
The AST is now translated into the IR. If the AST is constructed well, and IR is well chosen, then this should be fairly straightforward. Three address code might be an example of an IR.

Optimisation (back end):
The IR is now optimised to increase how quickly it runs, or decrease how much space it uses etc. Some optimisation are easier than others, and some are very complex! Often, the optimisation stage is so complex, that it could be a whole separate part of the compiler (a middle stage in-between the front and back ends).

Code generation (back end):
This phase is concerned with things such as register allocation (NP-Complete), instruction selection (pattern matching), instruction scheduling (NP-Complete) and more. Architecture specific information may be used for further optimisation here, and machine code is the output.

3 Lexical Analysis

So, lexical analysis is where we read the characters in the input and produce a sequence of tokens (i.e. tokenize the input). We want to do this in an automatic manner.

When translating from a programming language or a natural language, we need to map words to parts of speech. In programming languages, this is syntactic (as opposed to being idiosyncratic in natural languages), and is based on notation. Things like reserved words are very important in tagging a programming language.

To talk about lexical analysis, we should make some definitions:

See my computation notes (`COMP11212`) for more on CFG's

- A vocabulary (or alphabet) is some finite set of symbols.
- A String is a finite sequence of symbols from the vocabulary.
- A Context Free Grammar (CFG) is a 4-tuple; (S, N, T, P):
  - S: The starting symbol.
  - N: The non-terminal symbols.
  - T: The set of terminal symbols.
  - P: A set of production rules.
- A Language is any set of strings over a fixed vocabulary, or the set of terminal productions of a CFG.

Terminal symbols will usually start with a lower case letter, and non-terminal symbols will start with an upper case one.

We can use repeated substitution to derive *sentinel forms*.

Leftmost derivation is when the leftmost non-terminal symbol is expanded at each step. When we're recognising a valid sentence, we reverse this process.

If we have a knowledge about lexical analysis, we can avoid having to write a lexical analyser by hand, and can simplify the specification and implementation of a language. We need to specify a *lexical pattern* to derive tokens, which is essentially a CFG.

Some parts of this are easy, for example:

$$\text{WhiteSpace} \rightarrow \text{blank}|\text{tab}|\text{WhiteSpace blank}|\text{WhiteSpace}tab$$

Things like keywords, operators and comments are easy to use as well. However, some parts of languages are more complicated, such as identifiers and numbers. We want a notation that lets us go easily to an implementation.

Regular expressions are a way of specifying a regular language; they are formulas that represent a possibly infinite set of strings. A Regular Expression (RE) over the vocabulary V is defined as:

- ϵ is a RE denoting the empty set.
- If $a \in V$ then a denotes $\{a\}$.
- If r_1 and r_2 are RE's, then:
  - r_1^* denotes zero or more occurrences of r_1
  - $r_1 r_2$ denotes concatenation
  - $r_1|r_2$ denotes either or.
- Short-hands include $[a-d]$ which expands to $[a|b|c|d]$, r^+ for rr^* and $r?$ for $[r|\epsilon]$

Building a lexical analyser by hand may involve a function with lots of if/else statements as we try to classify each character one by one.

A different idea is to try and match the input to different regular expressions and use the one with the longest match. This would be linear in the number of regular expressions we have (we would need to do a fresh search for each).

We could study the regular expressions we have and try to automate the construction of a scanner. Some regular expressions can be converted into transition diagrams. For example, matching a register $(r(0-9)+)$ can be converted to the following:

The regex would accept the string if the transition diagram ends in an accept state after the string has been run through it. We can produce a transition table (one way of encoding automata in computers) and (once we've figured out how to produce the table, which we'll soon find out) run through the input string in linear time.

3.1 Deriving a Deterministic Finite Automata from a Regular Expression

Remember that a DFA is a special case of an NFA.

Every regular expression can be converted to a deterministic finite automaton (DFA), and DFA's can automate lexical analysis.

For example, the regular expression describing CPU registers (0-31) could be:

$$\text{Register} \to r((0|1|2)(\text{Digit}|\epsilon)|([4-9])|(3|30|31))$$

If we generated the DFA shown in Figure **??**, we could produce a transition table like:

State	r	$0,1$	2	3	$4\ldots9$
0	1	–	–	–	–
1	–	2	2	5	4
$2(fin)$	–	3	3	3	3
$3(fin)$	–	–	–	–	–
$4(fin)$	–	–	–	–	–
$5(fin)$	–	6	–	–	–
$6(fin)$	–	–	–	–	–

This is done by creating an NFA for each thing you can do in a regular expression (concatenation, either operator, star operator etc) and putting them together.

Converting a RE into an NFA is more direct, but its also a lot slower to parse (since NFA's can have many paths through them). Converting to a DFA is slower (and the resulting automata is slower), but lets us parse an input in linear time.

The idea is:

That's Ken Thompson, the guy who wrote unix.

1. Write down the RE for the input language
2. Convert it to an NFA (Thompson's construction)
3. Build a DFA to simulate the NFA (subset construction)
4. Shrink the DFA (Hopcroft's algorithm)

We have seen the subset construction algorithm before (in COMP11212), where it was called 'Algorithm One'. There is a rather compact but (I think) working implementation in `COMP36512/programs/nfa-dfa.py` on Github that you could look at.

Before we look at the algorithm itself, we need to look at two operations:

`move(states, a)` Returns all the states to which there is a transition from some state in `states` with the symbol `a`.

`e-closure(states)` All the states that are reachable using only epsilon transitions from any state in `states`.

The algorithm is:

```
dStates = e-closure(startState)
markedStates = []
dTable = {}
// While there are unmarked states
while (dStates - markedStates) is not []:
  t = (dStates - markedStates).get(0)
  markedStates.add(t)
  for symbol in alphabet:
    U = e-closure(move(t, symbol))
    if U in dStates:
      dStates.add(U)
    dTable(t, a) = U
```

This builds up a table (representing a DFA) by creating states representing all the states reachable with a specific symbol from a set of states within the NFA.

3.2 DFA minimisation; Hopcroft's algorithm

The main disadvantage of DFA's is that they can be rather large (when converting from an NFA to a DFA, the DFA could have worst case exponential number of states). As a result, Hopcroft's algorithm for minim sing the number of states in a DFA is good for our memory usage.

The idea behind the algorithm is to find groups of states where for each input symbol, every state in the group will have transitions exclusively within the group.

The algorithm is:

```
while the groups are not stable {
  for each group g {
    for each input symbol I {
      for each state s in g {
        if s(g) not in g {
          put s in its own group
        }
      }
    }
  }
}
```

3.3 Fast scanners

Now it is very easy to build a recogniser for the DFA, we can just get the transition table for the DFA, and read each character of the input, transitioning to the next state every time. Accepting and rejecting depends on whether we get to `eof` in a finishing state (accept)or we didn't/we made an invalid transition (reject).

However, table recognisers aren't as fast as they could be; each character requires one memory access, which is not very efficient, especially if the transition table is big and won't fit in a cache. Some automatic code generation tools will output a series of `goto` or `case` statements instead, since then a transition table isn't needed (the states are hardcoded into the program).

Poor language design can complicate analysis though. For example, in Fortran:

```
DO5i=1,25
  ...
5
```

Is a for-loop from 1 to 25. However, the following is also valid:

```
DO5i=1.25
  ...
5
```

This may not exactly be what happens, but it's the thought that counts.

The only difference is that there is a comma instead of a period, but now the compiler will have assigned the token `DO5i` a value of 1.25, and the 5 at the end will be a non-operation.

Similarly, in C++ having types within types can be hard since `myType<myType2<int>>` includes the symbols `>>`, which is the operator for writing to an output stream.

`lex` and `flex` are tools for generating lexical analysers.

4 Parsing languages

Not all languages can be described by regular expressions (as we have attempted to do so far). They cannot describe things like nested constructs (i.e. is the bracketing legal in a mathematical formula). Also, you can't use a terminal symbol in a regular expression before it has been fully defined (i.e. no recursion).

Chomsky's hierarchy of grammars tells us which grammars can describe other ones:

Phase structured:

Context sensitive:

Context free:

Regular:
Only rules of the form $A \rightarrow \epsilon$, $A \rightarrow a$, $A \rightarrow pB$ are allowed.

Context free syntax is expressed with a context free grammar, which is a 4-tuple $G = \{S, N, T, P\}$, where S is the start symbol, N are the non-terminal symbols, T are the terminal symbols and P are the production rules. We re- write rules (starting from the initial rule) in order to make them closer to a sentence we're trying to obtain. A sequence of re-writes leading to some desired output is called a *derivation*, and the process of finding a derivation for a specific sentence is called *parsing*.

A derivation is a sequence of steps from the initial rule to a parsed representation of the input. Different choices of steps can potentially lead to different derivations of the same input. Two derivations are of particular interest to us:

Leftmost derivation; at each step, replace the leftmost non-terminal.

Rightmost derivation; at each step, replace the rightmost non-terminal.

A parse tree is a graphical representation of a derivation that is independent of the order that the derivation rules were applied in. To construct a parse tree, start with the starting symbol (which is the root of the tree), then for each derivational step, add children nodes for each symbol on the right hand of the production rule to the node for the left side. After you've done this, the leaves of the tree will (should) read from left to right to form the input.

If we're parsing something like a mathematical formula, then we need to build the notion of precedence into our derivational rules so that we get the correct parsing for an input. To do this, we force the parser to recognise the high- precedence subexpressions first.

Grammars are ambiguous if they can possibly produce more than one parse tree for a sentence. This is the case if a grammar has more than one leftmost or rightmost derivation for a single sentinel form. An example is:

```
Stmt = if Expr then Stmt | if Expr then Stmt else Stmt | Expr
Expr = W | X | Y | Z
```

This could match the following in two ways:

```
if X then if Y then W else Z
```

In order to eliminate ambiguity (and also deal with precedence), we need to re- write the grammar:

```
Stmt = IfElse | IfNoElse
IfElse = if Expre then IfWithElse else IfWithElse
IfNoElse = if Expr then Stmt | if Expr then IfElse else IfNoElse
Expr = W | X | Y | Z
```

So, if there is ambiguity, then we should try and resolve it in the CFG (after all, we only have to do this once, instead of resolving at runtime every time!). Overloading (when you give multiple things the same name or structure, such as method overloading in Java) can create ambiguity that you can only resolve using context. In general, to solve ambiguity:

If the issue is context free, then re-write the CFG so that there is no ambiguity.

If the issue requires context to solve, then we need more information in order to continue. This is really a problem with the design of the language.

There are two broad classes of parsers; top down parsers and bottom up parsers:

Top-down parsers:

- Construct the root of the tree first, then construct the rest in pre-order, using a depth first search.
- Pick a production, and try to match the input. If you fail then use backtracking.
- Try to find a leftmost derivation for the input string where we scan from left to right.
- Some grammars are backtrack-free (called predictive parsing)

Bottom-up parsers:

- Construct the tree for an input but start at the leaves and work up towards the root.
- Use a left-to-right scan and try to construct a rightmost derivation in reverse.
- Handles a large number of grammars.

4.1 Top Down Recursive Descent Parsing

This is where you:

- Construct the root with the start symbol for the grammar
- While the leaves of the parse tree don't match the input:
  - Select a node in the tree labelled A.
  - Select a production rule for A and construct a child for each symbol on the right hand side of the rule.
  - When a terminal symbol is added to the fringe, but it doesn't match the fringe, then backtrack.

If you choose the right production step at each stage, then this algorithm is fast, if you don't then the runtime will increase due to backtracking.

A grammar is left-recursive if it has a non-terminal symbol A such that there is a derivation $A \to Aa$ for some string a. Such a grammar can cause an infinite look for recursive-descent parsers since they may output an string of infinite a's. To avoid this, we try to eliminate left recursion; i.e. replace $A \to Aa|b$ with $A \to bA\ A' \to aA'|\epsilon$.

The general algorithm for non-cyclic, non-ϵ-productive grammars is:

- Arrange the non-terminal symbols $A_1 \dots A_n$
- For $i = 1 \dots n$:
  - For $j = 1 \dots i - 1$:
    - Replace each production $A_i \to A_j\gamma$ with $A_I \to \delta_1\gamma| \dots |\delta_k\gamma$, where $A_j \to \delta_1| \dots |\delta_k$ are the current A_j productions.
    - Eliminate the left recursion among A_i.

4.1.1 Predictive parsing

So now we can make a top-down parser which will find the correct matching in finite time, but it employs backtracking when it picks the wrong initial rule (which slows everything down). If we look ahead in the input, we can use context to influence our decisions about what production rules to try next so we can attempt to reduce the amount we have to backtrack.

In general, we need an infinite amount of look-ahead to always make the right decision about what production rule to use next (but this is pretty much the same as parsing the rest of the string, right?). However, most context free grammars can be parsed with a limited look-ahead.

The basic idea is for any production $A \to a|b$, we want to have a distinct way of choosing the correct production to expand (either a or b).

We can do this for some grammars by defining a set called `FIRST`. This is the set of the first terminal symbols for each string derived from A. A very simple example is:

$$A \to B|C$$

$$B \to x|y$$

$$C \to m|n$$

$$FIRST(B) = \{x, y\}$$

$$FIRST(C) = \{m, n\}$$

If for a production rule $A \to X|Y$, $FIRST(X) \cap FIRST(Y) = \emptyset$, then we can implement the look-ahead with only one symbol. This is the LL property.

If a grammar does not have the LL(1) property, we can sometimes transform it so that it does:

- For each non-terminal A, find the longest prefix a common to two or more of its alternatives (e.g. B and C in $A \to B|C$).
- If $a \neq \epsilon$ then replace all the A productions $A \to ab_1|\ldots|ab_n|\gamma$ (where γ doesn't begin with a) with $A \to aZ|\gamma, Z \to b_1|\ldots b_n$.
- Repeat until we have no common prefixes.

Note that it is undecidable as to whether an LL(1) grammar exists for any CFG. It is also possible to parse non-recursively if we maintain a stack and determine what production rules to apply using a table.