CO221 - Compilers

6th January 2020 — Lecture

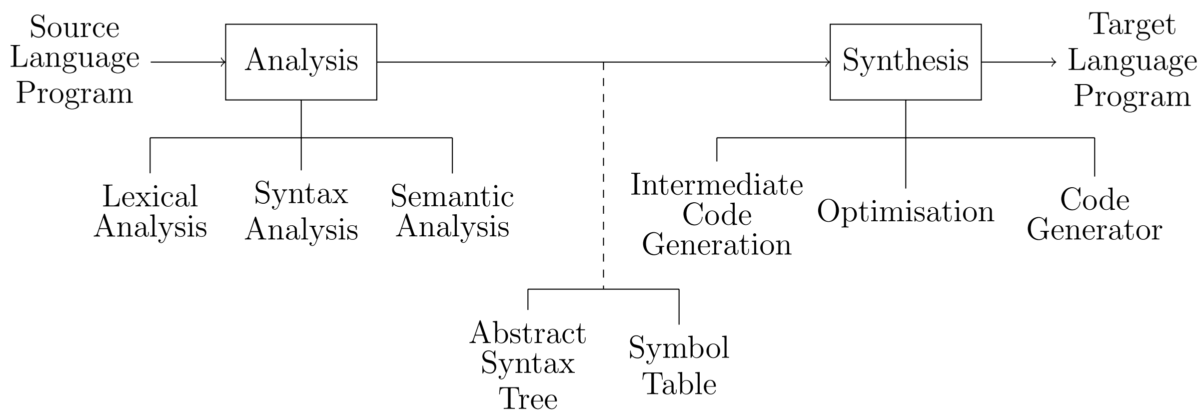
A compiler is a program which processes programs, including translating a program written in one language (usually higher level) to another programming language (usually in a lower level). In our course, the focus is to generate assembly code from the high level language. This translation goes between high level human concepts, and the data manipulation the machine performs.

Structure

The general structure of a compiler is as follows;

1. **input** takes in an input program in some language
2. **analysis** constructs an interal representation of the source structure
3. **synthesis** walks the representation to generate the output code
4. **output** creates an output in the target language

In more detail, it can be represented as follows;



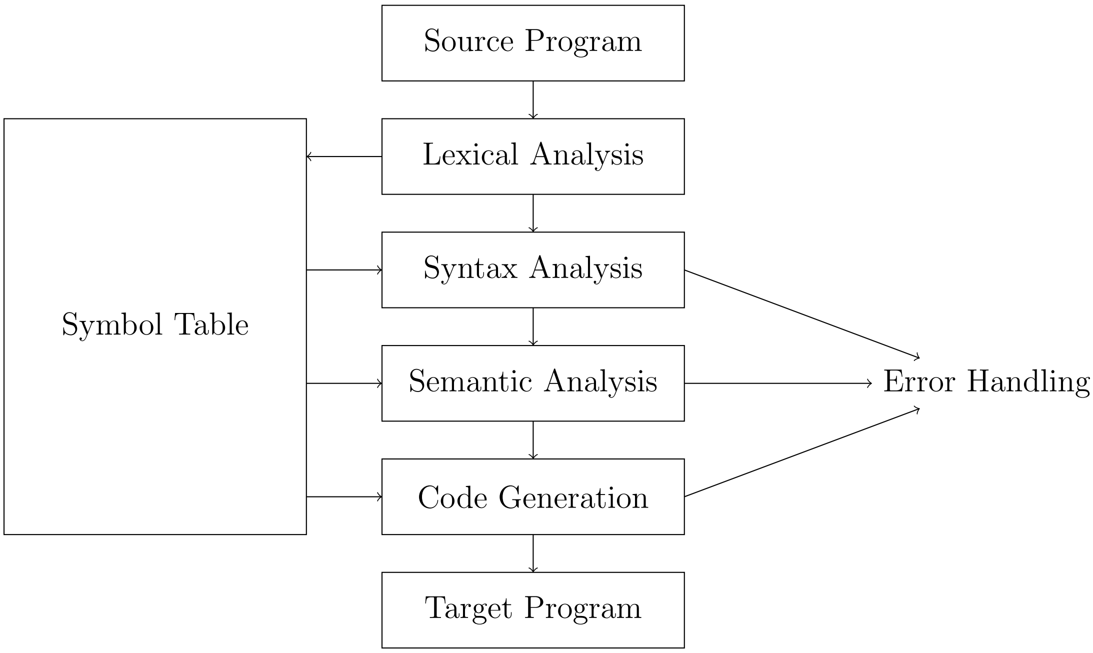
- **lexical analysis** looks at characters of input program, analyses which are keywords (such as converting `if`, and `while` to corresponding tokens), which are user defined words, and which are punctuation, etc.
- **syntax analysis** discovers structure of input
- **semantic analysis** checks that variables are declared before they are used, and that they are used consistently with their types etc.

Simple compilers go straight to code generation, but optimising compilers do several passes of intermediate code generation and optimisation.

The symbol table holds data on variables, such as types. Sometimes we need to know the type of the variable, in order to generate code for the variable, for example if we were to print a variable, it would need to generate different code for strings than it would need to do for integers. Scope rules are also needed.

Phases

Whether all of these phases are done in the order shown is a design choice. For example, lexical analysis and syntax analysis are often interleaved. This can be done when the syntax analysis stage needs the next symbol, and therefore the lexical analysis stage can be used.



Syntax Analysis

This is also known as parsing. Languages have a grammatical structure specified by grammatical rules in a **context-free grammar** such as BNF (**Backus-Naur Form**). The output of the analyser is a data structure which represents the program structure; an **abstract syntax tree**. The writer of the compiler must design the AST carefully such that it is easy to build, as well as easy to use by the code generator.

A language specification consists of the following;

- **syntax** grammatical structure

  in order to determine that a program is syntactically correct, one must determine how the rules were used to construct it
- **semantics** meaning

For example, we can encode the rules for a statement as follows (anything in quotes is a terminal), in BNF;

$$\text{stat} \to \text{'if' '(' exp ')' stat 'else' stat}$$

Each BNF production is a valid way for a non-terminal (LHS) to be expanded (RHS) into a combination of terminals and non-terminals. Only terminals can appear in the final results (they are lexical tokens).

To prove the following is a valid example of stat, we'd need to show that a can be derived from exp, and that both b and c can be dervied from stat.

$$\text{if (a) b else c}$$

Context-Free Grammars

Formally, a context-free grammar consists of the following four components;

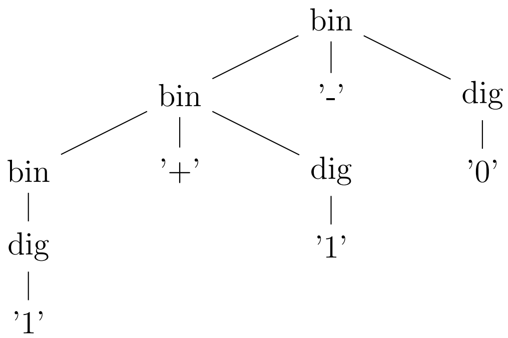
- S a non-terminal start symbol
- P a set of productions
- t a set of tokens (terminals)
- nt a set of non-terminals

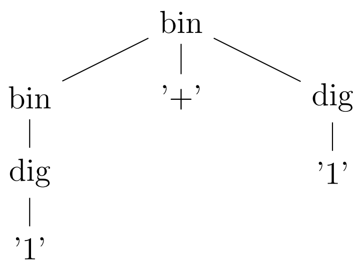
For example, consider the following BNF, and their associated components;

$$\begin{aligned} \text{bin} &\rightarrow \text{bin '+' dig} \mid \text{bin '-' dig} \mid \text{dig} \\ \text{dig} &\rightarrow \text{'0'} \mid \text{'1'} \\ t &= \{\text{'+'}, \text{'-'}, \text{'0'}, \text{'1'}\} \\ nt &= \{\text{bin}, \text{dig}\} \\ S &= \text{bin} \end{aligned}$$

A string of only terminals (**sentinal form**) can be derived using the grammar by beginning with the start symbol, and repeatedly replacing each non-terminal with the RHS from a corresponding production. We refer to the set of all sentinal forms derived from the start symbol as the **language** of a grammar.

We can prove that some string is in the language of a grammar by constructing a **parse tree**. For example, to prove that "1+1-0" $\in L(G)$, and "1+1" $\in L(G)$ we can use the following trees;

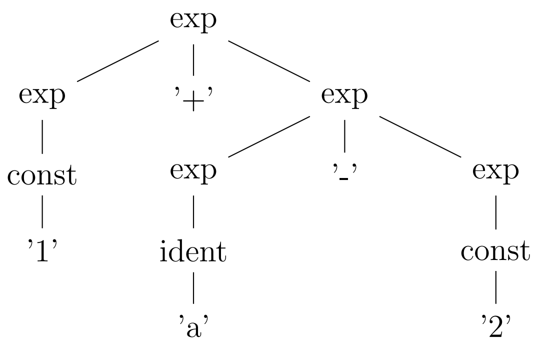


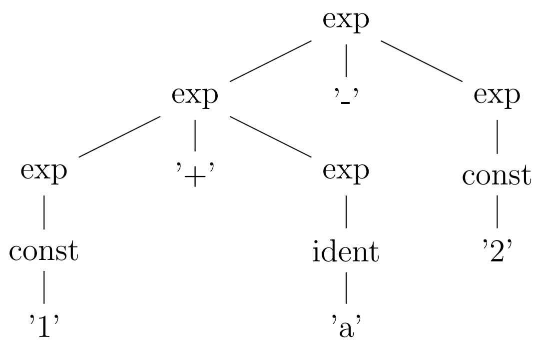


Ambiguity

A grammar is referred to as **ambiguous** if its language contains strings which can be generated in two different ways. Essentially, there exists some string in $L(G)$ which has two different parse trees. Consider string "1 + a - 3" in the following grammar, and the parse tree(s) associated;

$$\text{exp} \rightarrow \text{exp '+' exp} \mid \text{exp '-' exp} \mid \text{const} \mid \text{ident}$$





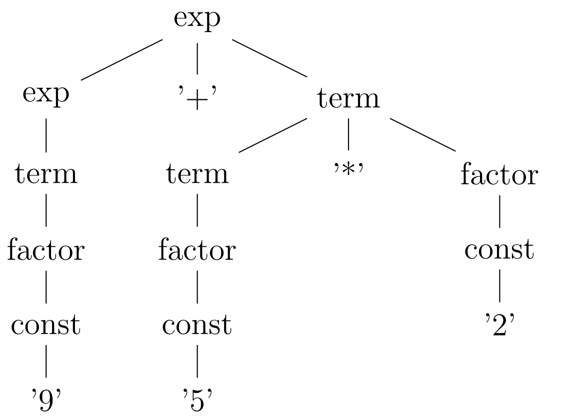
While the string is still valid, and in the language, our issue is with the ambiguity, as we want to generate a program uniquely. The reason our grammar is broken is due to the recursive use of the non-terminal exp on both sides, which means we're given a choice of which side to expand when generating.

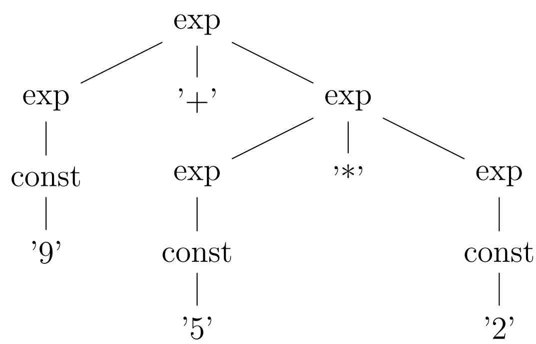
Associativity and Precedence

For our example language, we're using all left-associative operators. We also want to maintain that '*' and '/' have higher precedence than '+' and '-'. One way of doing this is to split the grammar into layers, by having seperate non-terminals for precedence levels. This method can be done with the following unambiguous grammar for arithmetic expressions;

$$\begin{aligned}
\text{exp} &\to \text{exp '+' term} \mid \text{exp '-' term} \mid \text{term} \\
\text{term} &\to \text{term '*' factor} \mid \text{term '/' factor} \mid \text{factor} \\
\text{factor} &\to \text{const} \mid \text{ident}
\end{aligned}$$

Now, we can unambiguously generate the parse tree (and thus the unique abstract syntax tree) for "9+5*2";





It's important to note that the **abstract** syntax tree doesn't need this in contrast, as only the parse tree needs it.

Parsers

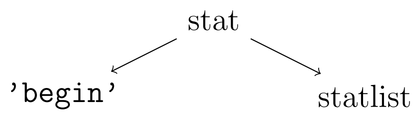
The parser checks that the input is grammatically correct, and builds an AST representing the structure. In general, there are two classes of parsing algorithms;

- **top-down / predictive** — we are using recursive descent
- **bottom-up** — also known as shift-reduce

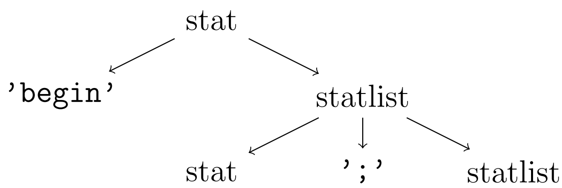
For this, we will use the input "`begin S; S; end`", with the following grammar;

stat → `'begin'` statlist | `'S'`
statlist → `'end'` | stat `';'` statlist

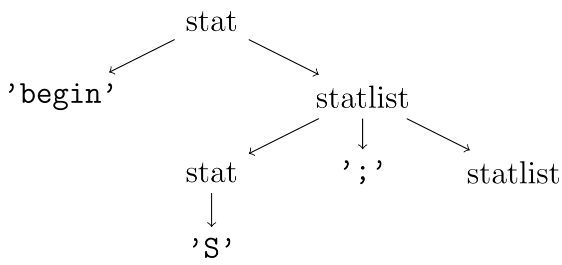
When we start top-down parsing, we start with the non-terminal stat. The first token we identify is the `'begin'`, thus our tree becomes the following;



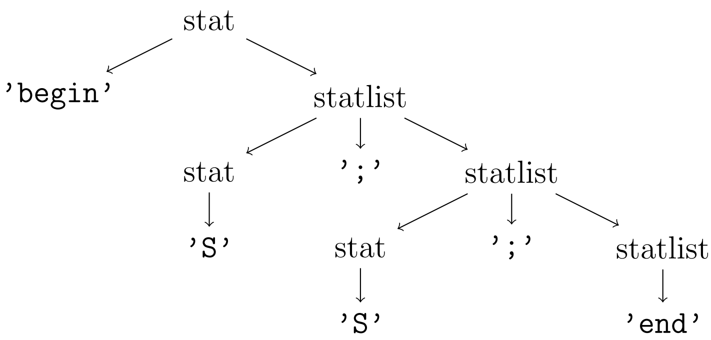
However, as the next symbol isn't the terminal `'end'`, we have to use an alternative. As we only have one alternative, we can predict it, and thus the tree becomes;



As the next symbols are the terminal `'S'`, and the terminal `';'`, we can tick them off, thus the tree becomes;



This process continues, until we reach the final tree;



On the other hand, bottom-up parsing tries to use all the RHSs (whereas top-down tries to match a non-terminal by trying each of the RHSs), and replaces it with a non-terminal, by using the production in reverse. Bottom-up succeeds when the whole input is replaced by the start symbol.

In general, we push the current symbol onto the stack (or the reduction if we can reduce it). For example, once we encounter `'S'`, we can reduce it to stat, and similarly once we encounter `'end'`, we can reduce it to statlist.

stack	current symbol	remaining tokens	S/R	note
	'begin'	'S' ';' 'S' ';' 'end'	S	nothing to do yet
'begin'	'S'	';' 'S' ';' 'end'	R	terminal for stat
'begin'	stat	';' 'S' ';' 'end'	S	no more work
'begin' stat	';'	'S' ';' 'end'	S	nothing to do yet
'begin' stat ';'	'S'	';' 'end'	R	terminal for stat
'begin' stat ';'	stat	';' 'end'	S	no more work
'begin' stat ';' stat	';'	'end'	S	nothing to do yet
'begin' stat ';' stat ';'	'end'		R	terminal for statlist
'begin' stat ';' stat ';'	statlist			
'begin' stat ';' stat ';'	statlist		R	match for statlist
'begin' stat ';'	statlist			
'begin' stat ';'	statlist		R	match for statlist
'begin'	statlist			
'begin'	statlist		R	match for stat
	stat			complete