Compilers and computer architecture

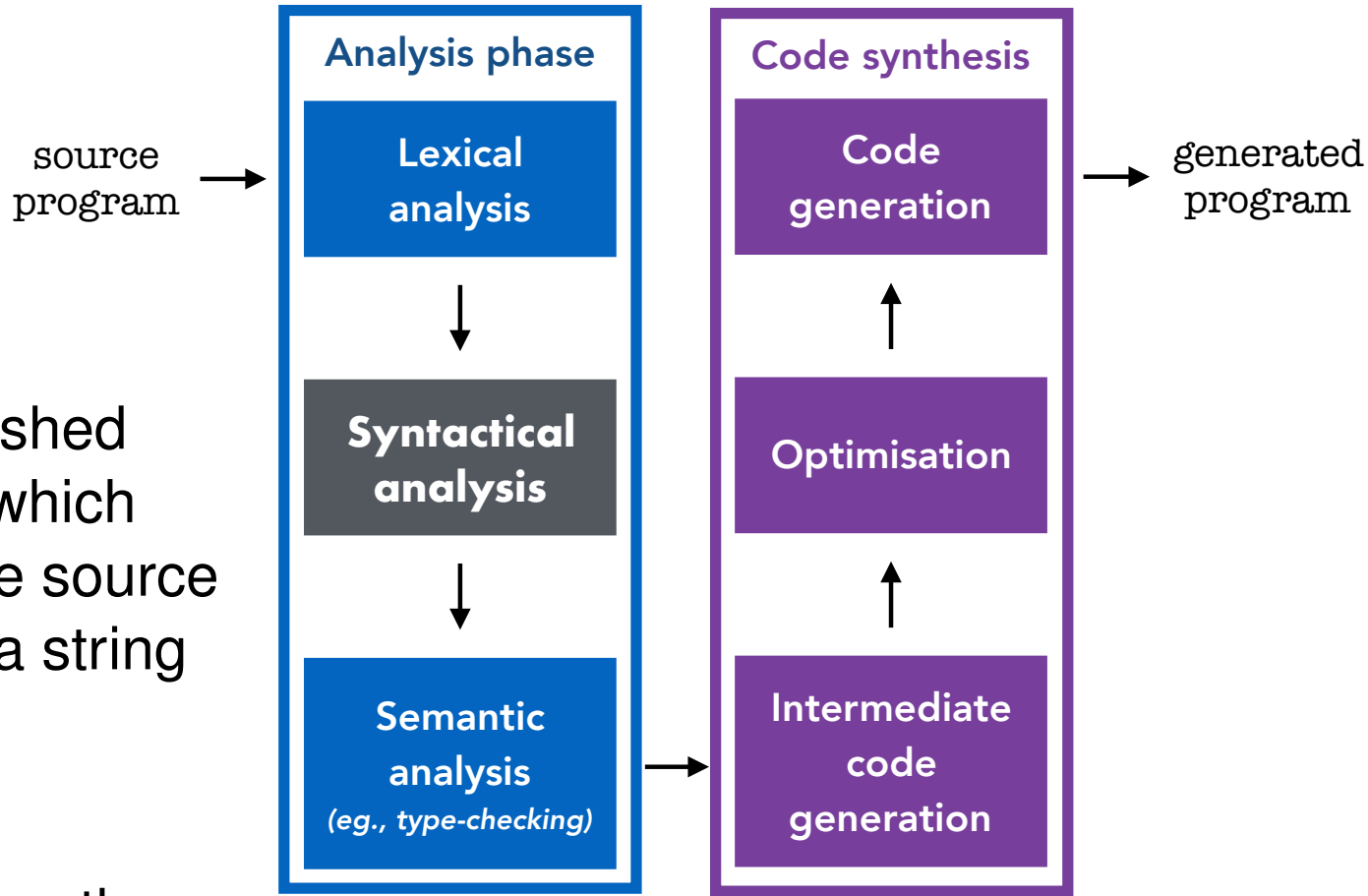
5. From strings to ASTs: parsing with context free grammars

Ian Mackie

Autumn 2022

Plan for this week

Remember the shape of compilers:



We have finished with lexing, which interprets the source program as a string of tokens.

We next **parse** the program — analysing its syntactical structure — to determine if it is well-formed.

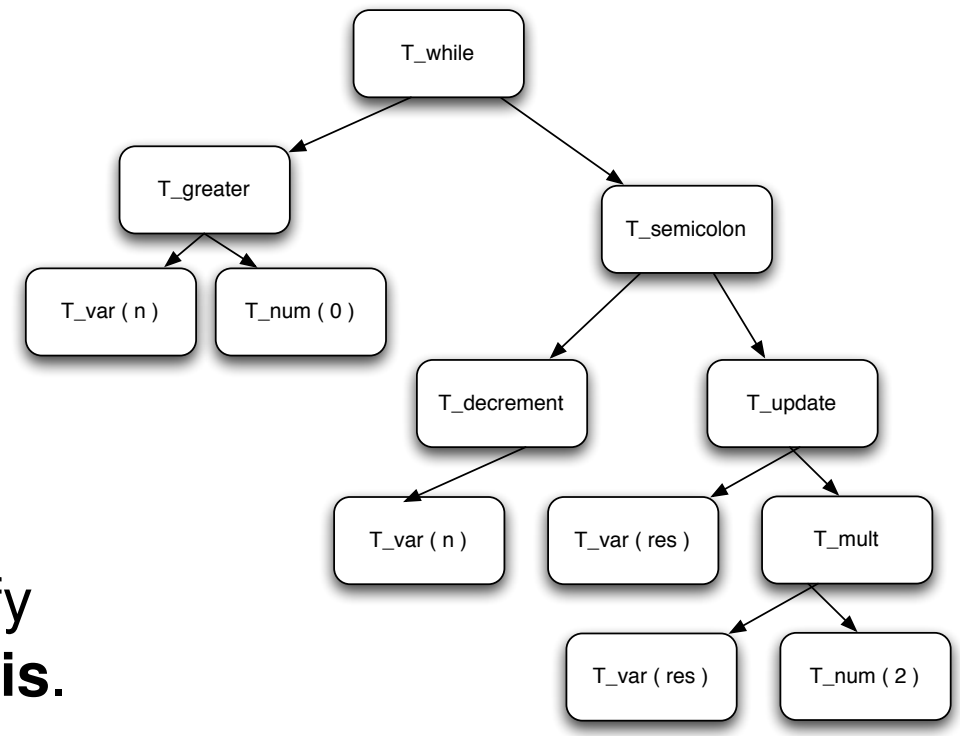
From strings to ASTs

We want to take a program given as a string and:

- Check if it's syntactically correct, *e.g.*:
  - Is every open bracket later closed?
  - Is it a sequence of valid declarations / statements?
- Produce an AST, to facilitate efficient **code generation**.

Recall, we want to go from the representation on the left, to that on the right:

```
while (n > 0) {
  n--;
  res *= 2;
}
```



Lexing is a good first step, but is mostly useful to simplify the step of **syntactic analysis**.

Why we need syntactic analysis

Lexical analysis breaks the input into the basic pieces of a computer program. That doesn't guarantee that the input makes sense as a computer program.

Question. Are the following inputs lexically 'correct' for Java?

- `for ( int i = 0 ; i < 5 ; i ++ ) n += i ;`
  keyword keyword iden num iden num iden iden iden
- `for ( int i = 0 ; ] ; { for {`
  keyword keyword iden num keyword
- `if at first , you dont succeed , try , try again`
  keyword iden iden iden iden iden keyword keyword iden

From lexing to parsing

Lexing — breaks input into **atoms** which could form a valid program, if they combine into grammatical structures

Parsing — assembles these atoms into **structures** which could form a valid program, if they have a semantically coherent interpretation



Exhibit A. Lack of apparent structure

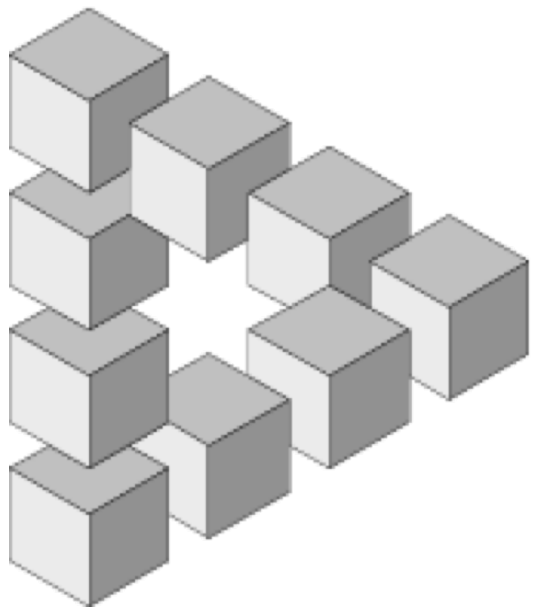


Exhibit B. Lack of *globally coherent* structure

From lexing to parsing

- Syntactic structure does not care about differences in variable names, constants, types, ... so, the following are syntactically equivalent:

```
if   abs(x-y) < 5  then P else Q

if floor(a*a) == 2 then P else Q
```

- The lexer makes this easier by hiding irrelevant data (variable names, specific constants, *etc.*), putting the **syntactic type** of each token front and centre:

```
T_if  T_iden T_lparen T_iden T_optor ...
```

Getting started with syntactic analysis

As with lexical analysis, we have to describe

- a **specification** for syntactically correct programs;
- a way to **check** that an input is syntactically correct
  — and: to generate an output (the AST)
  that is useful to the next compiler stage.

To accomplish this, it would be helpful to be able to use regular expressions / finite state automata again (or something similar).

Let's deal with specification first.

- How can we describe a program structure?
- Can we use regular expressions for this purpose?

Regular expressions aren't enough
to express program syntax

We want to be able to express program structures which have nested, balanced **opening** / **closing** tokens, such as:

```
for (i=0; i < (a+b)/2; i++) {
   if (f(i) == 0) { /* ... */ }
}
```

Unfortunately, REs (and FSAs) can't express the constraint of parentheses, braces, etc. being balanced.

Question. Why not?

Answer. Because FSAs can't count

— and because every RE can be decided by an FSA,
no RE can express a language which requires counting.

What do you mean, “FSAs can’t count”?

Finite State Automata can’t express the constraint of parentheses, braces, etc. being balanced

— at some point they can’t keep reliable count of the number of open brackets.

- Each FSA has only a fixed number of states (some $n > 0$).
- What happens if we have more than n open brackets before we hit a closing bracket?
  - When we reach the n^{th} open bracket, we run out of states (or we did at some earlier point)
  - When reading some bracket, say the m^{th} ofne (for some $m \leqslant n$), we must return to a state which we visited before, say when reading the k^{th} bracket (for some $k < m$)
  - The FSA cannot distinguish between k open brackets, or m open brackets
- That is: the FSA loses count!

Bracket-like structures in programming languages

The inability of FSAs (and regular expressions) to count is a problem for expressing practical programming languages, as nesting bracket-like constructs are very common: e.g.,

```
begin ... end

do ... while

if ( ... ) then { ... } else { ... }

3 + ( 3 - (x + 6) )
```

To allow a computer to process these constructs, we need a language formalism that *can* 'count' in this way — or, which can at least **do matching** (openings vs. closings).

The solution: **context free grammars (CFGs)**.

Context free grammars

CFGs are a generalisation of regular expressions, that are ideal for describing such recursive and nested structures.

Example. A way to express non-empty strings over $\{$ **(**, **)** $\}$ which have perfectly balanced brackets:

$$\textsc{BalancedBrackets} \to \texttt{(}\ \texttt{)}$$

$$\textsc{BalancedBrackets} \to \texttt{(}\ \textsc{BalancedBrackets}\ \texttt{)}$$

$$\textsc{BalancedBrackets} \to \textsc{BalancedBrackets}\ \ \textsc{BalancedBrackets}$$

One way to read this: recursive rules for ways to **generate** strings of balanced brackets — e.g.,

$$\begin{aligned}\textsc{BalancedBrackets} &\to \texttt{(}\ \textsc{BalancedBrackets}\ \texttt{)}\\ &\to \texttt{(}\ \textsc{BalancedBrackets}\ \ \textsc{BalancedBrackets}\ \texttt{)}\\ &\to \texttt{(}\ \texttt{(}\ \texttt{)}\ \texttt{(}\ \textsc{BalancedBrackets}\ \texttt{)}\ \texttt{)} \to \texttt{(}\ \texttt{(}\ \texttt{)}\ \texttt{(}\ \texttt{(}\ \texttt{)}\ \texttt{)}\ \texttt{)}\end{aligned}$$

Context free grammars

A **context-free grammar** is a tuple $(A, V, Init, R)$ such that:

- A is a finite set (**alphabet** / **terminals**)
- $V \neq \varnothing$ is a finite set s.t. $V \cap A = \varnothing$ (**variables** / **non-terminals**)
- $Init \in V$ (the **start symbol**)
- $R \subseteq V \times (A \cup V)^*$ is a finite set (**productions** of the CFG)

  — *i.e.*, R is a relation on variables $X \in V$, and strings s containing both variables and alphabet characters.

  We often write "$X \to s$", instead of $(X, s) \in R$.

Alternative terms: productions $\to$ "reductions" | "transitions"

start symbol $\to$ "initial variable"

Context free grammars

The balanced brackets example:
this is a CFG (A, V, S, R) such that

- $A = \{ \texttt{(}, \texttt{)} \}$ and $V = \{\textsc{BalancedBrackets}\}$.
- The initial variable is just $S = \textsc{BalancedBrackets}$.
- R contains the three reductions:

$$S \to \texttt{(}\ S\ \texttt{)}$$
$$S \to S\ S$$
$$S \to \texttt{(}\ \texttt{)}$$

— This still relies on the informal description of how CFGs describe strings through “possible generation rules”.

To make this more precise, we need to formally define what the language of a CFG is.

Reductions in CFGs

The key idea is simple:
use reductions to **replace the occurrences of variables in the context of longer strings** which may contain both alphabet characters and variables.

- Given a string $s \in (A \cup V)^*$, any occurrence in s of a variable X can be replaced by the string $r = r_1 ... r_n \in (A \cup V)^*$, provided that $X \to r$ is a valid reduction.

Example. If we have a reduction $S \to \mathtt{a}\, X\, \mathtt{b}$, then we can rewrite the string

$$\mathtt{a\,a}\, S\, \mathtt{b\,b}$$

to

$$\mathtt{a\,a\,a}\, X\, \mathtt{b\,b\,b}$$

How do we start? — With the initial variable.

When does it stop? — When there are no more variables to replace.

The language accepted by a CFG

If you like formal definitions . . . Let $\mathcal{G} = (A, V, S, R)$ be a CFG.

- For arbitrary strings $s, t \in (V \cup A)^*$, we define the *one-step reduction relation* "$s \Rightarrow t$" between strings $s, t \in (V \cup A)^*$, where $s \Rightarrow t$ if and only if
  - $s = p\,X\,r$ where $X \in V$ and $p, r \in (V \cup A)^*$.
  - $t = p\;q\;r$ for some $q \in (V \cup A)^*$.
  - There is a reduction $X \to q$ in R.
- We write $s \Rightarrow^* t$ if there is a chain of reductions,

$$s \Rightarrow s_1 \Rightarrow \cdots \Rightarrow s_n \Rightarrow t,$$

for some strings $s_1, \ldots, s_n \in (V \cup A)^*$, where $n \geqslant 0$.

- The **language accepted by** $\mathcal{G}$, written $\mathrm{lang}(\mathcal{G})$, is

$$\mathrm{lang}(\mathcal{G}) \stackrel{def}{=} \{t \in A^* \mid S \Rightarrow^* t\}$$

A chain of reductions $S \Rightarrow s_1 \Rightarrow \cdots \Rightarrow s_n \Rightarrow t$ is called a **derivation** of t.

The language accepted by a CFG

The language of a CFG (A, V, S, R) is the set of all strings in A^* that can be produced by a chain of one-step reductions, from the initial variable S.

Let's do this with the CFG $\mathcal{G}$ of non-empty strings of balanced brackets:

- $A = \{ \texttt{(}, \texttt{)} \}$ and $V = \{S\}$
- R contains the reductions

$$S \to \texttt{(}\ S\ \texttt{)} \qquad S \to S\ S \qquad S \to \texttt{(}\ \texttt{)}$$

Then we can show again that "(( ) (( )))" is in lang($\mathcal{G}$):

$$S \Rightarrow \underline{\texttt{(}\ S\ \texttt{)}} \Rightarrow \texttt{(}\ \underline{S\ S}\ \texttt{)} \Rightarrow \texttt{(}\ \underline{\texttt{(}\ \texttt{)}}\ S\ \texttt{)} \Rightarrow \texttt{(}\ \texttt{(}\ \texttt{)}\ \underline{\texttt{(}\ S\ \texttt{)}}\ \texttt{)} \Rightarrow \texttt{(}\ \texttt{(}\ \texttt{)}\ \texttt{(}\ \underline{\texttt{(}\ \texttt{)}}\ \texttt{)}\ \texttt{)}$$

Question: can CFGs “count”?

Answer: just enough for our needs.

- CFGs don’t **literally** ‘count’, or keep any record of numbers of characters.
- CFGs **do** allow characters to be generated in opposing pairs (or more complicated patterns) — more **performing matching**, than counting.

This allows for a wide range of expression, and more than enough to express well-structured programs in the usual way.

Remark. There are languages that a CFG cannot express: *e.g.*,

$$\left\{ \mathtt{a}^k \, \mathtt{b}^k \, \mathtt{c}^k \;\middle|\; k \geqslant 0 \right\}$$

over $A = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, where $s^k = \overbrace{ss \cdots s}^{k \text{ times}}$. (A language which could *really* “count”, would be able to do this!)

Infinite derivations?

Many CFGs allow infinite *reductions*. For example, in the grammar for non-empty strings of balanced brackets:

$$
\begin{aligned}
S &\to (S) \\
&\to ((S)) \\
&\to (((S))) \\
&\to ((((S)))) \\
&\to (((((S))))) \\
&\to ((((((S)))))) \\
&\quad\vdots
\end{aligned}
$$

But a **derivation**, of a string $t \in A^*$, must eliminate every variable at some point —

- so it can't involve an infinite reduction;
- and it can't yield an infinitely long string.

Simple arithmetic expressions

For example:

EXPR $\rightarrow$ NUM | EXPR **+** EXPR | EXPR **−** EXPR | EXPR ***** EXPR | **(** EXPR **)**

NUM $\rightarrow$ *(a number)*

The language contains:

- 7
- $7 * 4$
- $7 * 4 + 222$
- $7 * (4 + 222)$ ...

That is: all arithmetic expressions involving addition, subtraction, and multiplication of integer constants.

A well-known context free grammar

What language does the following grammar describe?

$$R \longrightarrow \emptyset \mid \varepsilon \mid \text{'c'} \mid R \mid R \mid RR \mid R^* \mid (R)$$

Regular expressions over an alphabet A, are themselves the language of a CFG! (But not the language of another RE.)

Question: Since regular expressions are a special case of CFGs, why not do both lexing and parsing using only CFGs?

Answer: In principle we could — but:

- Lexing is an easier problem $\Longrightarrow$ admits simpler solutions
- REs are more limited $\Longrightarrow$ admit faster algorithms

Some conventions regarding CFGs

Here is a collection of conventions for making CFGs more readable. You will find them a lot when programming languages are discussed.

- Instead of $N \to r_1$, $N \to r_2$, $N \to r_3$, $\vdots$ we may write $N \to r_1 \mid r_2 \mid r_3 \mid \vdots$

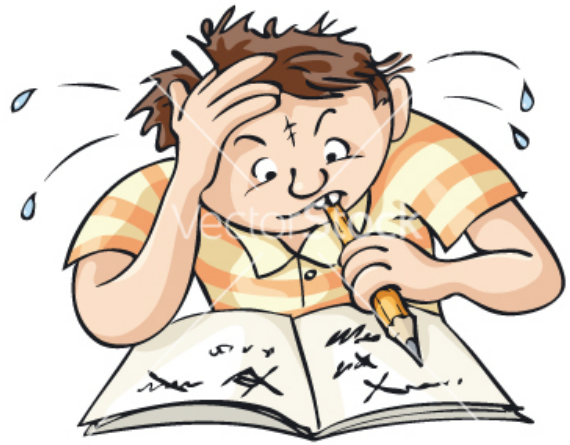
  — or alternatively $N \to r_1 \mid r_2 \mid r_3 \mid \cdots$.

- Many write $::=$ instead of $\to$ for reduction rules.
- It is helpful to clearly distinguish variables and alphabet characters, *e.g.*:
  - CAPITALISED variables, lower-case alphabet (or vice versa).
  - **Bold** variables, normal-weight alphabet (or vice versa).
  - Variables written in ⟨angle-brackets⟩, the alphabet isn't.

Example: Java grammar

Let's look at the CFG for a real language:

`docs.oracle.com/javase/specs/jls/se13/html/jls-19.html`



A slightly different format, using

variable **:**
  string_1
  string_2
  ⋮

instead of

$$\begin{aligned} \textit{variable} &\rightarrow \textit{string_1} \\ &\mid \textit{string_2} \\ &\mid \ \vdots \end{aligned}$$

(Other different shorthand as well — but not difficult to follow.)

CFGs, what's next?

Recall what we need, to solve the problem of parsing —

- a **specification** for syntactically correct programs;
- a way to **check** that an input is syntactically correct, and to generate the AST.

CFGs generalise REs, and allow us to specify the grammar for programming languages . . . but we also want:

1. An algorithm that decides whether a token list is in the language of the grammar, or not.
2. An algorithm that (for syntactically valid inputs) converts the list of tokens into an AST.

The key idea to solving both problems in one go:
the **parse tree**.