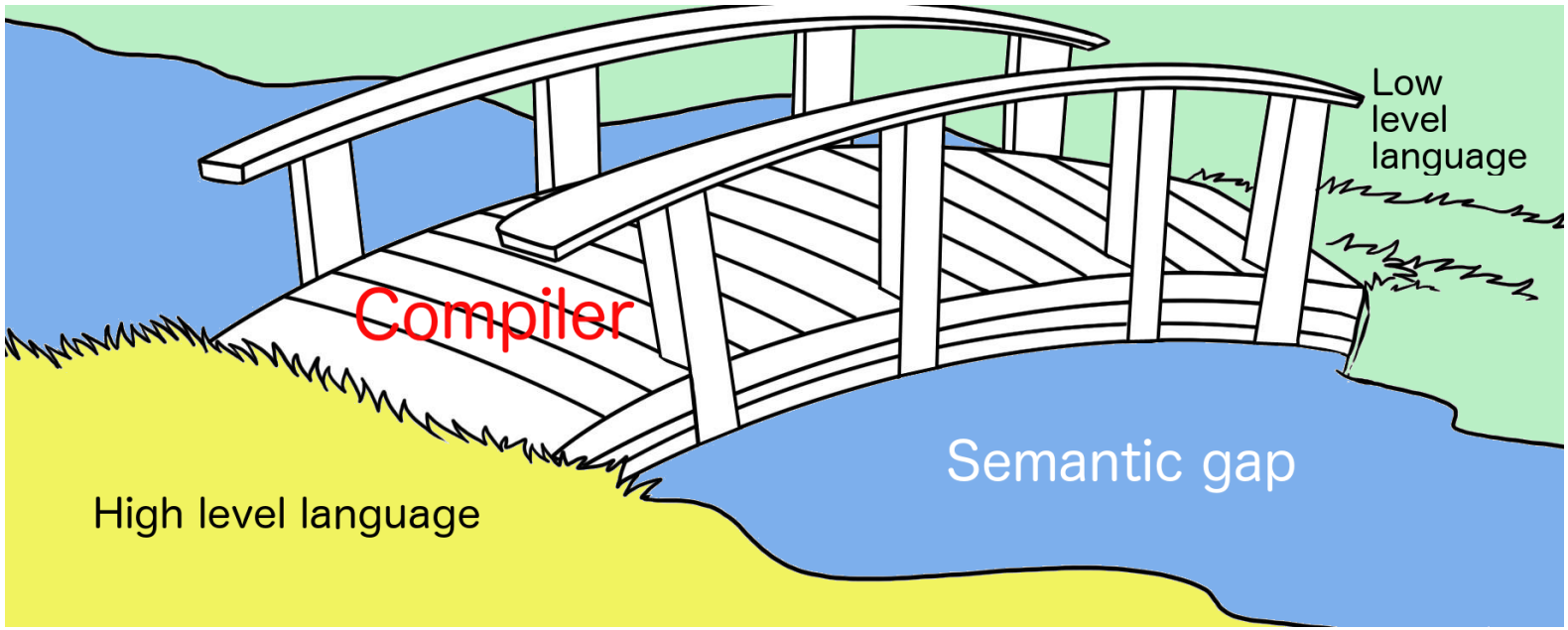
Compilers and computer architecture: Parsing

Martin Berger [1]

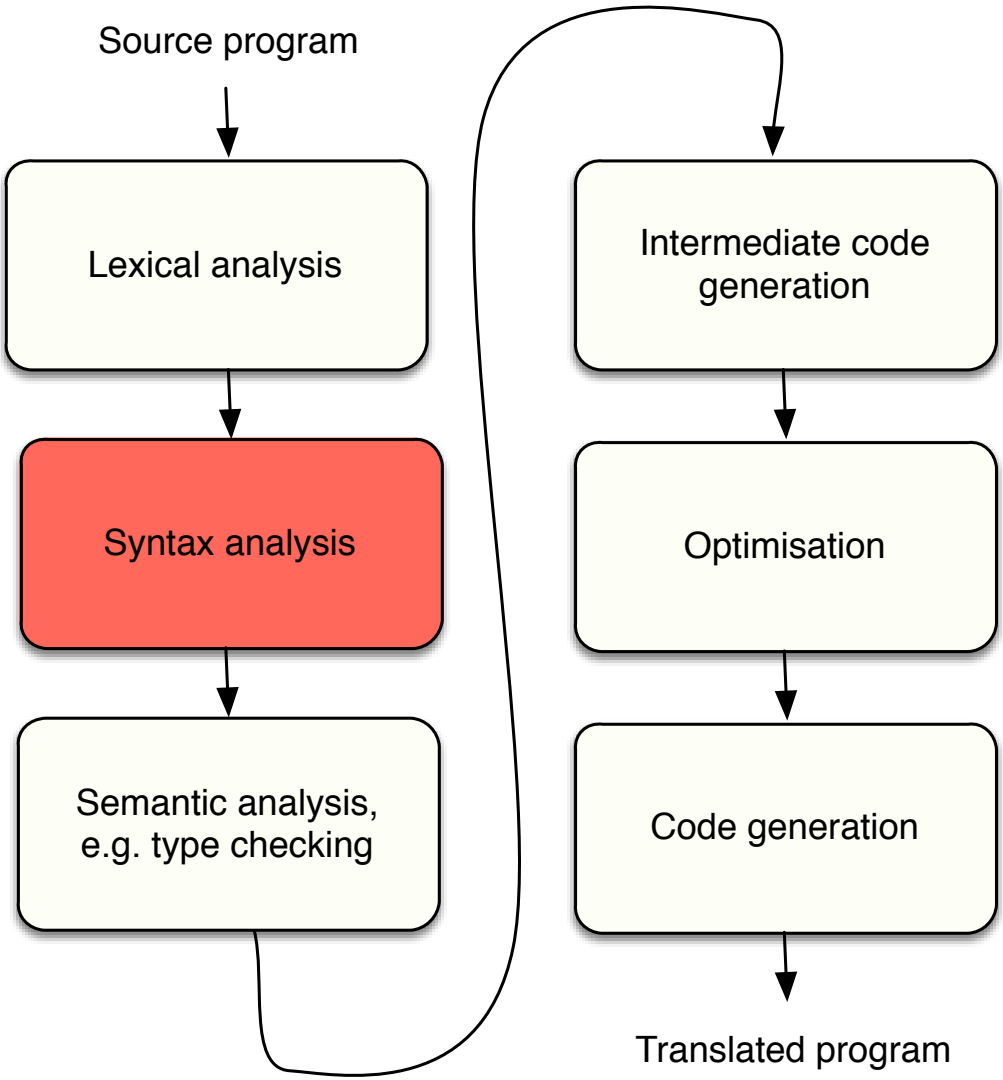
October 2019

[1] Email: M.F.Berger@sussex.ac.uk, Office hours: Wed 12-13 in Chi-2R312

Recall the function of compilers



Recall we are discussing parsing



Key steps

Remember we need:

- **Specify** the syntax of the language.
- Have an **algorithm** that decides the language and returns an AST.

$$\frac{\text{regular expression/FSA}}{\text{FSA}} \quad = \quad \frac{\mathit{CFG}}{???}$$

Key steps

- CFGs as mechanism to specify syntax with recursive structure $\mathcal{G} = (A, V, I, \rightarrow)$.
  - V variables, A alphabet.
  - $I \in V$ initial variable.
  - Transitions $X \rightarrow \sigma$ where $X \in A$, $\sigma \in (A \cup V)^*$
  - If $X \rightarrow \sigma$ then $\alpha X \beta \Rightarrow \alpha \sigma \beta$.
  - The **language** of $\mathcal{G}$ is $\{\sigma \in V^* \mid I \Rightarrow \cdots \Rightarrow \sigma\}$.
  - We call each $I \Rightarrow \cdots \Rightarrow \sigma$ a **derivation**.

Parser

A parser (in its simplest form) for a CFG $\mathcal{G}$ takes as input a string/token-list, and returns true/false depending on whether the string is in the language of $\mathcal{G}$ or not, and, at the same time, build an AST representing the structure of the input.

Key concepts: derivations and parse trees. The latter can be seen as a 2D representation of the former. It is very close to ASTs (but contains some redundant information like brackets). When constructing the AST we drop those, and only keep stuff that's needed for later phases of the compiler.

Parser

There are two approaches to parsers:

- **Top down** or predictive (we will study the recursive descent algorithm).
- **Bottom up** (also known as shift-reduce algorithms)

Two approaches to parsers

- Top down.
  - Good: conceptually easy, can be hand-written, powerful.
  - Bad: Can be slow. Can't deal with left-recursing CFGs (see later), but left-recursive grammars can be converted automatically to non-left-recursive CFGs.
- Bottom up.
  - Good: Fast, can deal with all (non-ambiguous) CFGs.
  - Bad: Complicated, hard to write by hand.

We'll look at top down

Top down parsing: intuition for the decision problem

You look at the current string and go through all rules starting with a variable in the string (say leftmost) if the input can be used with one of the rules, if it matches.

If there's a matching rule, recursively process the rest of the string. (To start, we must rewrite the initial variable.)

If you 'consume' the whole string, you are finished.

If you get stuck, backtrack.

If you have exhaused all rules without success, reject the string.

Example top down parsing

$$
\begin{array}{lcl}
P & \rightarrow & \textit{begin } Q \\
P & \rightarrow & \textit{prog} \\
Q & \rightarrow & \textit{end} \\
Q & \rightarrow & P \ ; \ Q
\end{array}
$$

Let P be the initial variable. Note: grammar not ambiguous!

Example string: `begin prog; prog; end`

Slogan for top-down parsing: "Starting from the initial variable, search for a rule which rewrites the variables to yield characters from the alphabet **consistent with the input**. As you rewrite, make a note (= AST) of which rules you applied how!"

This is what we've done in the examples so far when drawing parse trees.

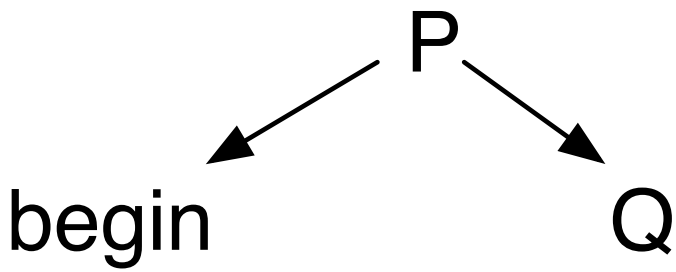
Example top down parsing

P

CODE: begin prog; prog; end

PARSED:

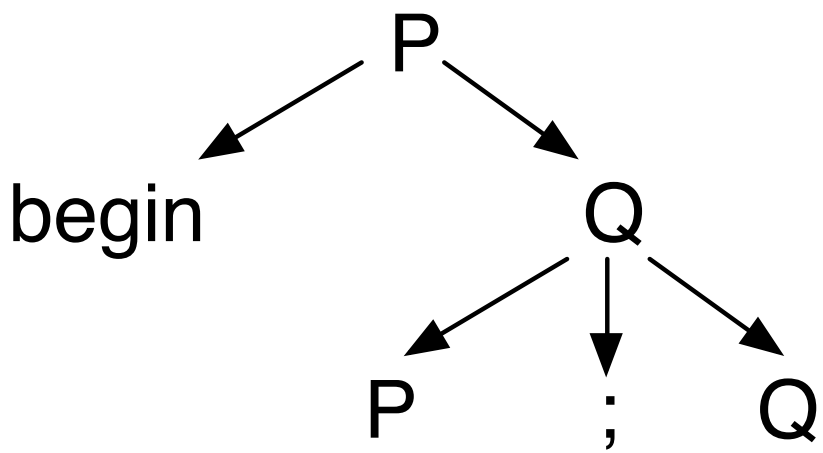
Example top down parsing



CODE: prog; prog; end

PARSED: begin

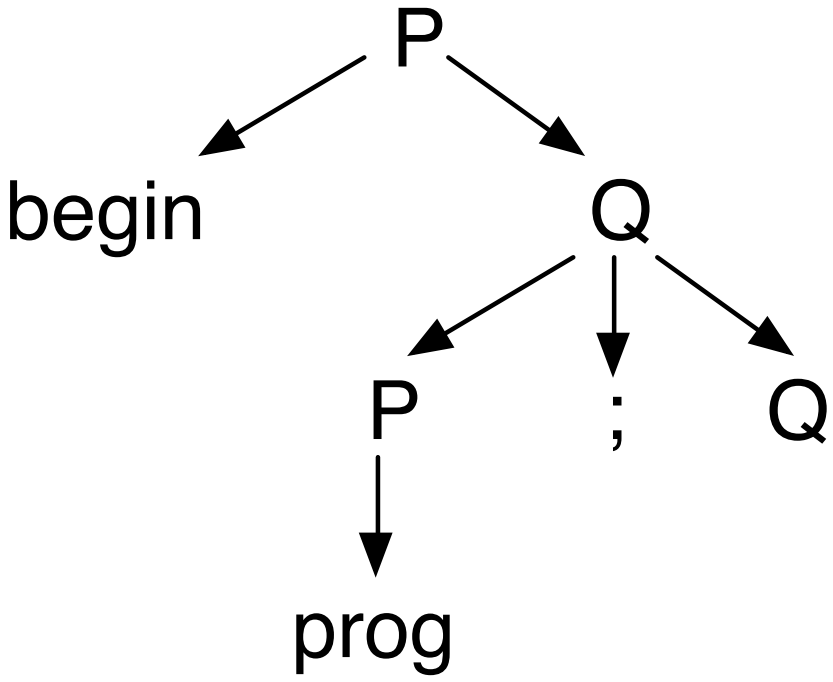
Example top down parsing



CODE: prog; prog; end

PARSED: begin

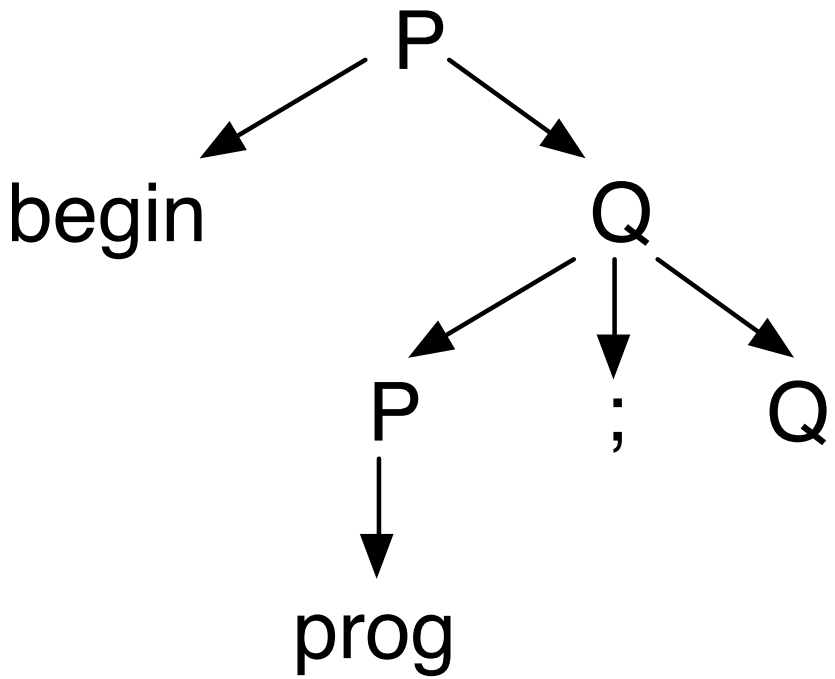
Example top down parsing



CODE: ; prog; end

PARSED: begin prog

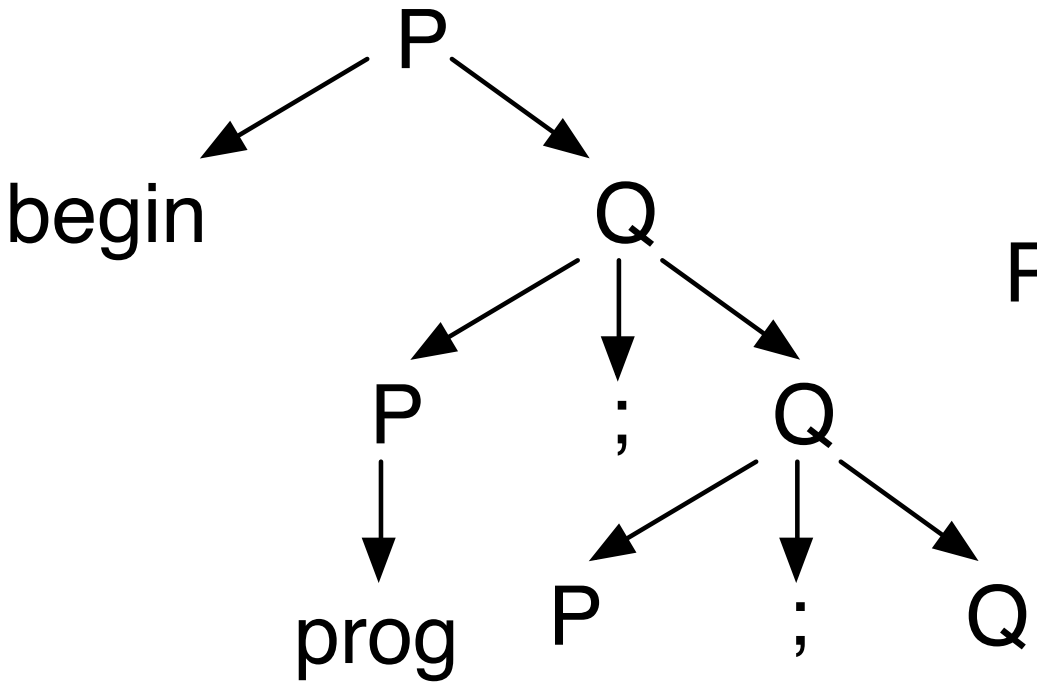
Example top down parsing



CODE: prog; end

PARSED: begin prog;

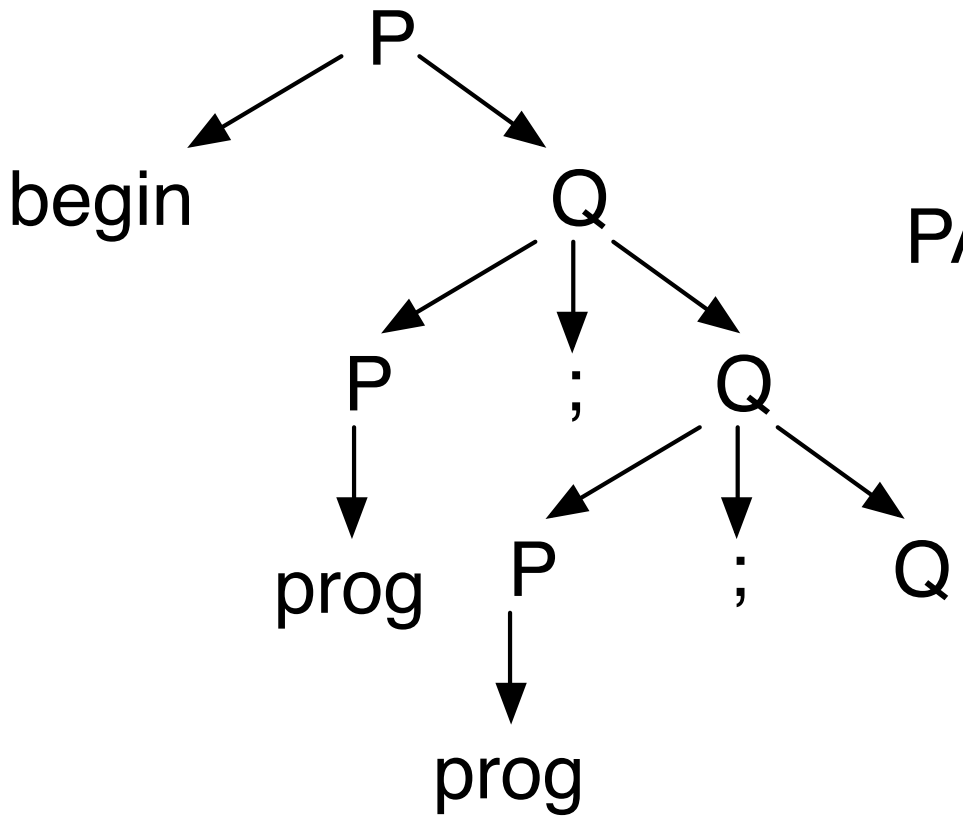
Example top down parsing



CODE: prog; end

PARSED: begin prog;

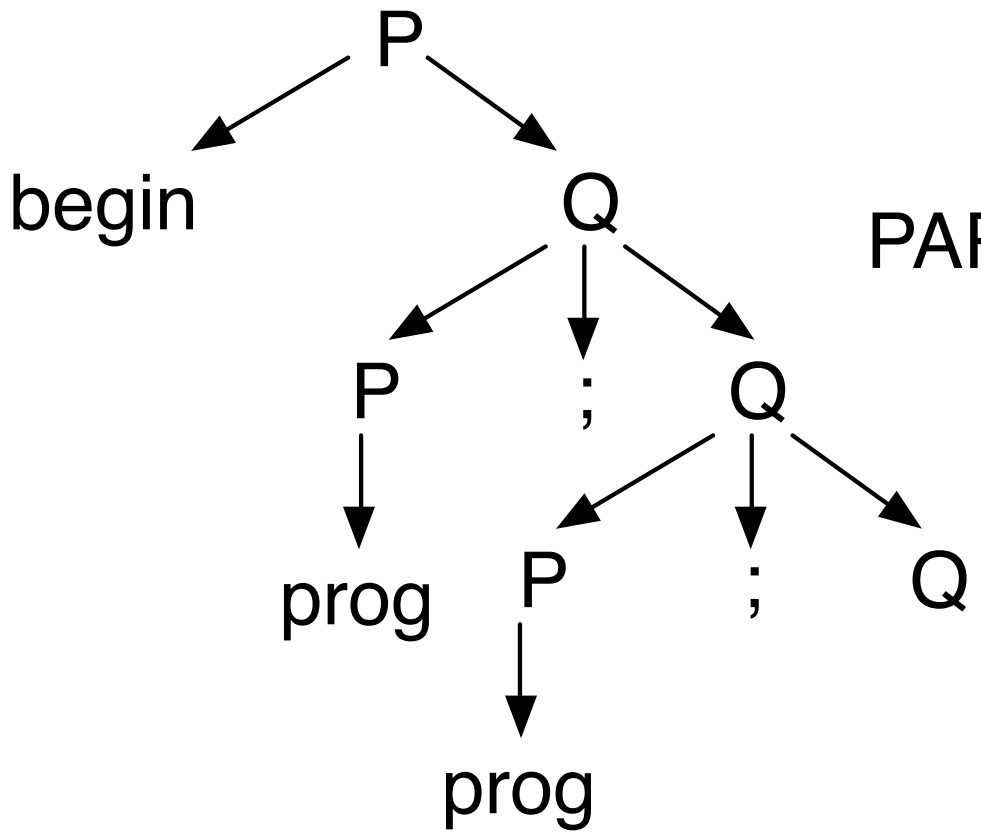
Example top down parsing



CODE: ; end

PARSED: begin prog; prog

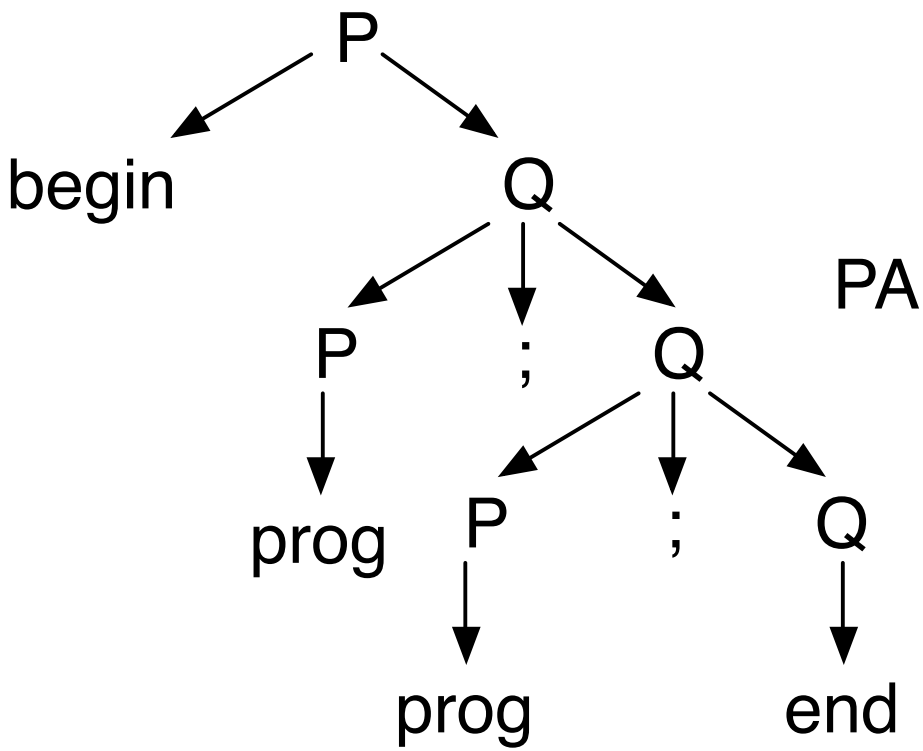
Example top down parsing



CODE: end

PARSED: begin prog; prog;

Example top down parsing



CODE:

PARSED: begin prog; prog; end

Summary of approach we used

- We start with the full input, and the initial parse tree (corresponds to the initial variable). We choose to work left-most (meaning we always rewrite the left-most variable first)
- Once, the parsing process is going, we assume the current input is derived from leftmost variable (let's call it X)
- Examine each alternative transition for X, say $X \rightarrow S$
- Compare **first (or more) unmatched** input token with S
  - If a matching transition is found (e.g. *begin*) use it to rewrite X, and remove the matched stuff from the input string
  - Repeat, using next input token to determine the transition to be used for the next variable
  - If no match, try a different transition. If nothing matches, reject input
- At each step, one of the transitions was chosen, and used from left-to-right, to replace a variable in the parse tree by a RHS.

Let's turn this into pseudo-code!

Top down parsing pseudo code

Recall our grammar.

$$
\begin{array}{lcl}
P & \rightarrow & \textit{begin } Q \\
P & \rightarrow & \textit{prog} \\
Q & \rightarrow & \textit{end} \\
Q & \rightarrow & P \ ; \ Q
\end{array}
$$

First we define our tokens in pseudo-code (for more complicated languages, tokens may carry additional information).

```
interface Token
   class T_begin () implements Token
   class T_end () implements Token
   class T_prog () implements Token
   class T_semicolon () implements Token
```

Top down parsing pseudo code

Recall our grammar.

$$
\begin{array}{lcl}
P & \rightarrow & \textit{begin}\ Q \\
P & \rightarrow & \textit{prog} \\
Q & \rightarrow & \textit{end} \\
Q & \rightarrow & P\ ;\ Q
\end{array}
$$

We use methods, one for each variable (can be programmed in other ways)

```
def parseQ( tl : List [Token] ) = ...
def parseP( tl : List [Token] ) = ...
```

Top down parsing pseudo code

We use methods, one for each variable (can be programmed in other ways)

```
def parseQ( tl : List [Token] ) = ...
def parseP( tl : List [Token] ) = ...
```

Each method does the following:

- Consume (eat) as much of the input as possible according to grammar for the variable (e.g. `parseQ` according to rules for *Q*).
- Indicate if no input can be parsed.
- If some input can be parsed, but not all, return the leftovers (= input not parsed), I'm ignoring the AST for now!

So a parse is successful exactly when all input is consumed. Then the input is in the language of the CFG.

Parsing P transitions

Recall our grammar.

$$
\begin{array}{lcl}
P & \rightarrow & \textit{begin } Q \\
P & \rightarrow & \textit{prog} \\
Q & \rightarrow & \textit{end} \\
Q & \rightarrow & P \ ; \ Q
\end{array}
$$

```
def parseP ( tl : List [ Token ] ) =
  if tl is of form
     T_begin :: rest then parseQ ( rest )
  else if tl is of form
     T_prog :: rest then rest
  else "rule doesnt apply"
```

Here `x::l` is short for a list with first element `x` and rest `l`.

Parsing Q transitions

Recall our grammar.

$$
\begin{array}{lcl}
P & \rightarrow & \textit{begin } Q \\
P & \rightarrow & \textit{prog} \\
Q & \rightarrow & \textit{end} \\
Q & \rightarrow & P \; ; \; Q
\end{array}
$$

```
def parseQ ( tl : List [ Token ] ) =
  if tl is of form
    T_end :: rest then rest
  else {
      let tl2 = parseP ( tl )
      if tl2 is of form
        T_semicolon :: rest2 then parseQ ( rest2 )
      else "rule doesnt apply"
```

In other words: if we have a terminal (token), we remove it from the input. If we have a variable, we call the associated parser.

Parsing P & Q transitions

That's it. No further code needed. That was simple. Usage:

```
t = List ( T_begin, T_prog, T_semicolon, T_end )

try {
  let result = parseP ( t )
  if ( result.size <= 0 )
      println ( "success" )
  else
      println ( "failure" ) }
catch {
  case e : Exception => println ( "failure" ) }
```

Top down parsing

This was simple. The approach can be refined massively, leading to **combinator parsing**, where you can more or less write a grammar, and it's a valid Haskell or Scala program. In Java, this is difficult due to lacking expressivity. Combinator parsers are extremely elegant in my opinion (albeit too slow for parsing large input).

However, we've omitted various issues.

- Doing something (e.g. constructing an AST) during parsing.
- Left-recursion.

Constructing an AST during parsing

Usually we don't just want to decide if an input string is in the language defined by the ambient CFG. Instead we want to build up something, e.g. an AST. This is quite easy. Here's a pseudo-code example for the grammar we've been looking at.

```
interface Token
   class T_begin implements Token
   class T_end implements Token
   class T_prog ( s : String ) implements Token
   class T_semicolon implements Token

interface AST
   class EmptyAST implements AST
   class ProgAST implements AST
   class SeqAST ( lhs : AST, rhs : AST ) implements AST
```

Constructing an AST during parsing

```
interface AST
   class EmptyAST implements AST
   class ProgAST implements AST
   class SeqAST ( lhs : AST, rhs : AST ) implements AST
```

Note: no ASTs corresponding to begin/end. This is bracketing, so it's implicit in AST structure anyway. (Why?)

Other options are possible.

Constructing an AST during parsing

We use options `None` and `Some(...)` to indicate success or absence of success in parsing. (See `java.util.Optional` in Java.) In case of success, we return `ast`, the AST that has been constructed and `rest`, the remaining tokens:

```
Some ( ( ast, rest ) )
```

Otherwise we return `None`. Many languages offer this, in Java you can use an interface `Optional` with implentations `None` and `Some( ... )` to do this. You can write a small auxiliary class to implement tuples like (x, y).

Optional.java

Constructing an AST during parsing

We also use the following abbreviation for the return type of our parse.

```
type Result = Optional[ Pair[ AST, List[ Token ] ] ]
```

Constructing an AST during parsing

Recall our grammar.

$$
\begin{array}{lcl}
P & \rightarrow & \textit{begin}\ Q \\
P & \rightarrow & \textit{prog} \\
Q & \rightarrow & \textit{end} \\
Q & \rightarrow & P\ ;\ Q
\end{array}
$$

Now we parse P like this:

```
def parseP ( tl : List [ Token ] ) : Result =
  if tl is of form
    T_begin :: rest then parseQ ( rest )
  else if tl is of form
    T_prog :: rest then
          Some ( ProgAST (), rest ) )
  else None }
```

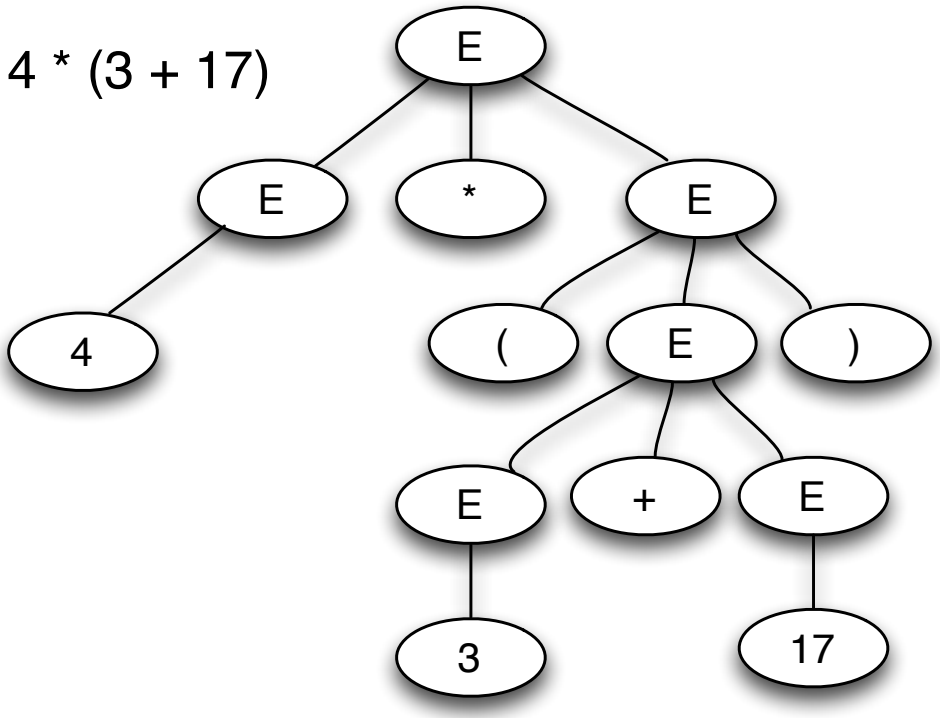
Note that the `then` clause is simple here because we have chosen **not** to represent `begin` / `end` by an explicit AST.

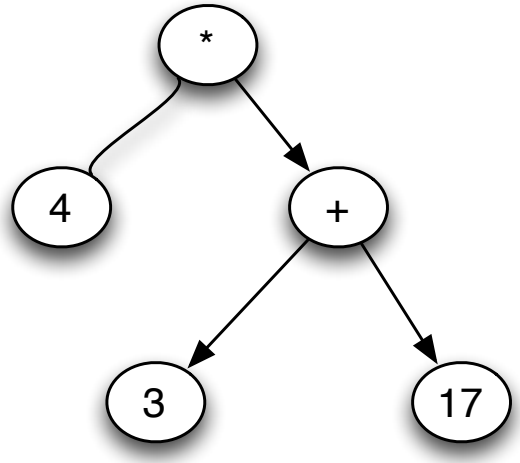
Constructing an AST during parsing

$$Q \rightarrow \mathit{end}$$
$$Q \rightarrow P\,;\,Q$$

```
def parseQ ( tl : List [ Token ] ) : Result =
  if tl is of form
    T_end :: rest then Some( ( EmptyAST, rest ) )
  else
    if parseP ( tl ) is of form
      None then None
      Some( ( astL, restL ) ) then
        if restL is of form
          T_semicolon :: restLL then
            if parseQ ( restLL ) is of form
              None then None
              Some( ( astR, rest2 ) ) then
                Some(SeqAST(astL, astR), rest2)) }}
        else None
    else None
```

Question: what's the difference between parse trees and ASTs?





Parse trees contain redundant information, e.g. brackets. Moreover, subsequent stages don't care about e.g. CFG variable names, so we can drop them.

ASTs convey the essence of the parse tree.

But you can always use the parse tree as AST.

Left-recursion

Recall our grammar.

$$
\begin{array}{lcl}
P & \rightarrow & \textit{begin } Q \\
P & \rightarrow & \textit{prog} \\
Q & \rightarrow & \textit{end} \\
Q & \rightarrow & P \ ; \ Q
\end{array}
$$

and its implementation in a simple top-down parser:

```
def parseP( tl ) = {
  if tl is of the form
    T_begin :: rest then parseQ ( rest )
    T_prog :: rest then rest
  else "no matching transition"
```

Key idea: each occurrence of a variable such as Q gives rise to a recursive call to a parser for that variable *parseQ*.

Left-recursion

Let's look at a grammar for expressions

$$E \rightarrow E + E \mid E * E \mid -E \mid (E) \mid 0 \mid 1 \mid \ldots$$

and its implementation in a simple top-down parser:

Left-recursion

$$E \rightarrow E + E \mid E * E \mid -E \mid (E) \mid 0 \mid 1 \mid \ldots$$

```
def parseE ( tl : List [ Token ] ) = {
  let tryAdd = parseE ( parse+ ( parseE ( tl ) ) )
  if ( parsing successful ... )
     tryAdd
  else
    let tryMul = parseE ( parse* ( parseE ( tl ) ) )
    if ( parsing successful ... )
      tryMul
    else if tl is of form
       T_leftBracket :: rest then ...
       T_minus :: rest then parseE ( rest )
       T_int ( n ) :: rest then rest
   ...
   else {
      ``[parseE] no matching transition'' )
```

Left-recursion

```
def parseE ( tl : List [ Token ] ) = {
  if tl is of form
     ...
  then {
    let tryAdd = parseE ( parse+ ( parseE ( tl ) ) )
    if ( tryAdd.size > 0 )
      tryAdd
    else {
      ...
```

The parser doesn't terminate!

$$\texttt{parseE}(2 + 3) \rightarrow \texttt{parseE}(2 + 3) \rightarrow \ldots$$

Left-recursion

The problem is that the grammar

$$E \rightarrow E + E \mid E * E \mid -E \mid (E) \mid 0 \mid 1 \mid \ldots$$

is **left-recursive**! Unlike

$$\begin{array}{lcl} P & \rightarrow & \mathit{begin}\ Q \\ P & \rightarrow & \mathit{prog} \\ Q & \rightarrow & \mathit{end} \\ Q & \rightarrow & P\ ;\ Q \end{array}$$

Why? because 'eating' `begin` shortens the argument for the recursive call.

Left-recursion

More generally, a grammar is **left-recursive** if we can find a variable N and a string σ such that

$$N \to \cdots \to N\sigma$$

So a grammar like

$$\begin{array}{lcl} P & \to & Q\ \mathit{hello} \\ Q & \to & P\ \mathit{world} \\ Q & \to & \mathit{end} \end{array}$$

is also left-recursive

Removing left-recursion

Good news! It is possible algorithmically to remove left-recursion from a grammar by introducing new variables. Here is an example.

$$R \rightarrow R\,\texttt{woof} \mid \texttt{baaa}$$

What is the language of this CFG? Strings of the form

$$\texttt{baaa woof woof ... woof}$$

We can rewrite this as:

$$\begin{aligned} R &\rightarrow \texttt{baaa}\,Q \\ Q &\rightarrow \texttt{woof}\,Q \mid \epsilon \end{aligned}$$

Here Q is a new variable, and ϵ the empty string. Now every 'recursive call' needs to 'chew off' an initial terminal. Hence recursion terminates.

Removing left-recursion

A more non-trivial example with two instances of left-recursion.

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid E - T \mid T \\
T & \rightarrow & T * F \mid T / F \mid F \\
F & \rightarrow & 0 \mid 1 \mid \ldots
\end{array}
\quad \overset{\textit{rewrite}}{\Rightarrow} \quad
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +T\ E' \mid -\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *F\ E' \mid /F\ T' \mid \epsilon \\
F & \rightarrow & 0 \mid 1 \mid \ldots
\end{array}
$$

With this new grammar, a top-down parser will

- terminate (because all left-recursion has been removed)
- backtrack on some (more) inputs

Removing left-recursion

Every left-recursive grammar $\mathcal{G}$ can be **mechanically** transformed into a grammar $\mathcal{G}'$ that is not left-recursive and accepts the same language.

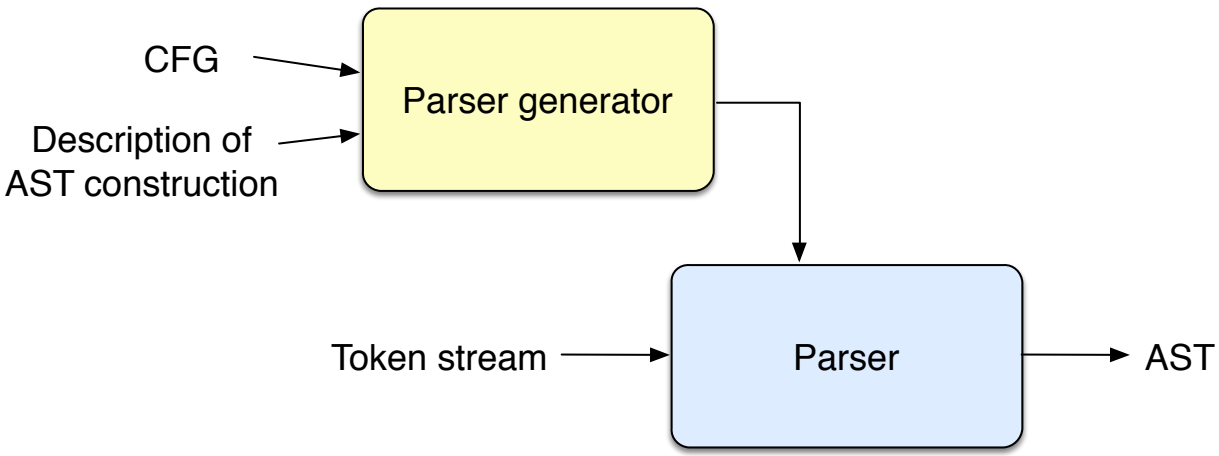
It's a fun and enlightening exercise to write implement this algorithm.

Parser and lexer generators

Great news: the construction of a top-down (and also bottom-up) parser is **completely mechanical** given a CFG.

Parser and lexer generators

Lexers can be generated by **lexer generators** (e.g. JFlex). Parsers can be generated by **parser generators** (e.g. JavaCC, CUPS, Yacc, Bison). Lexers and parsers generated by such tools are likely to be better (e.g. faster, better error handling) than hand-written parsers. Writing a lexer or parser by hand is typically much more time consuming than using a generator.



Parser generators usually provide precedence declarations to handle ambiguity (or they produce parsers that return all possible parse trees). They also handle left-recursion. They are available for most programming languages.

Using parser generators

There are many parser (and lexer) generators. To use them properly, you will have to read the manual. Often the input is something like this:

```
preamble

---------- boundary ----------

grammarRule action
...
grammarRule action
```

In the `preamble` you are setting up the parser, e.g. saying that the name and type of the parsing function to be produced should be, what the type of the input is, what the initial variable is etc.

Using parser generators

With each rule

```
grammarRule action
```

we must specify what should happen when we encounter input that can be parsed in accordance with the rule. For example a rule $E \rightarrow E + E$ could be handled something like this

```
E ::= E TokenPlus E { return new ASTPlus($1,$2); }
```

Here `E ::= E TokenPlus E` is the rendering of the rule $E \rightarrow E + E$ in the language of the generator. The red stuff is the action in the target language (e.g. Java).

The `$1 $2` are variables of the parser generator language and allow us recursively to access the results from parsing the left E (with `$1`) and right E (using `$2`) in $E + E$. Note that `$1 $2` will not occur in the generated Java.

Different generators may use different syntax.

Using parser generators

So an grammar/action like

```
E ::= E TokenPlus E { return new ASTPlus($1,$2); }
```

says the following: whenever the input matches $E + E$ then return a fresh object of class `ASTPlus`, and the constructor gets as first argument whatever matches the left E and as second argument whatever fits the right E.

So the input $3 + 4$ might lead to the parser an object generated like so.

```
new ASTPlus ( new IntLiteral ( 3 ),
              new IntLiteral ( 4 ) )
```

(Here we assume that parsing an integer yields objects of class `IntLiteral`.)

Using parser generators

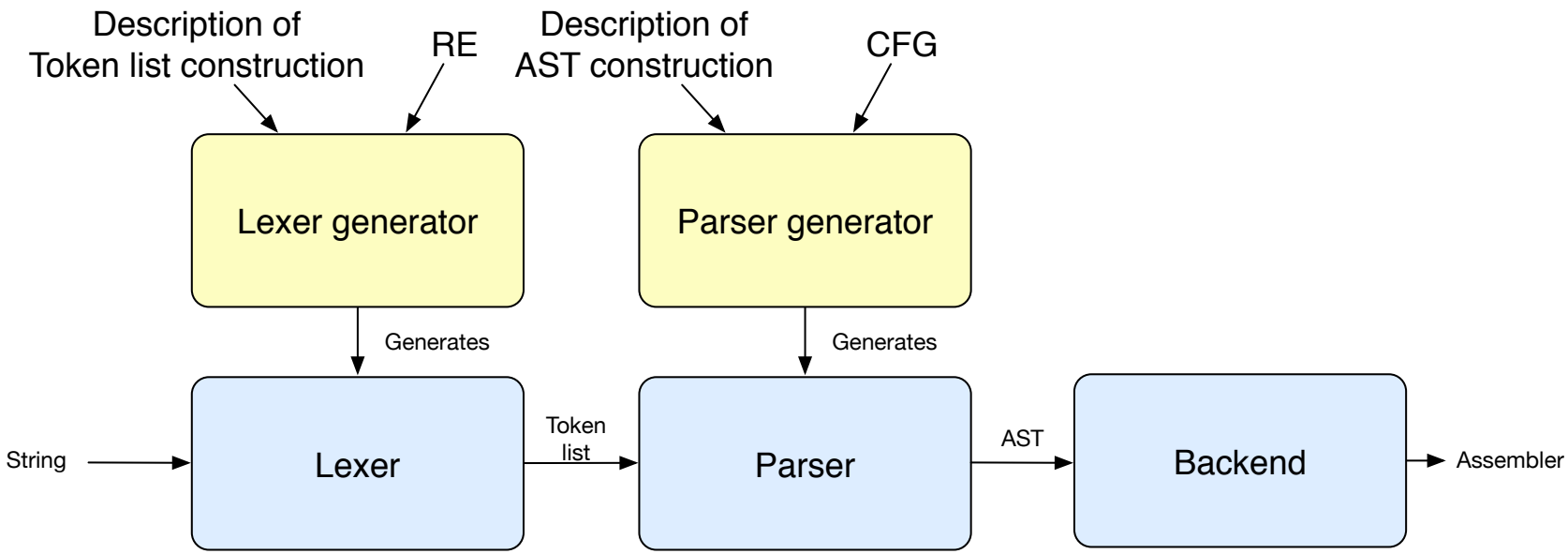
Note that you can embed **arbitrary** code in an action:

```
E ::= E TokenPlus E { print( "Let's go to Mars" ) }
```

The parser generator will simply generate a parser and add the action to handle the result of of successful parses. Note that parser generators typically don't check the action for syntatic correctness, instead it simply embeds the actions in the code it produces. That means the generated parse may not compile, or may produce non-sense. (And likewise for lexer generators.)

Using parser generators

It is the programmer's responibility to compile the code generated by the parser generator and integrate it with the rest of the compiler, i.e. feed the lexer's output to the generated parser, and feed the generated perser's output to the compiler's backend.



Using lexer and parser generators makes it easier to change a lexical or syntactic specification in contrast with changing a hand-written lexer / parser.

Conclusion

We use CFGs to specify the syntax of programming languages (after giving the lexical details using regular expressions/FSAs).

Parsers for a CFG are algorithms that decide if an input string is in the language of the CFG, and at the same time can build up ASTs.

Ambiguity means some string can be parsed in more than one way, this must be avoided in some way or other.

Parsers are either top-down (easy to write but require absence of left recursion) or bottom-up.

It's usually best to use lexer / parser generators to build lexers / parsers.

The theory of formal languages (like CFGs) is deep and beautiful.

The material in the textbooks

- Introduction to parsing
  - EaC Chapter 1
  - Dragon book Ch.s 1 and 2.
  - Appel & Palsberg Ch. 1
- General grammar issues, top-down parsing
  - EaC Chapter 3 sections 3.1-3.3
  - Dragon Book pp.42, pp.60 and Ch. 4
  - Appel & Palsberg Ch. 3.
- Parser-generators
  - EaC Section 3.5
  - Dragon Book Ch. 4 pp.287
  - Appel & Palsberg Ch. 3 pp.68