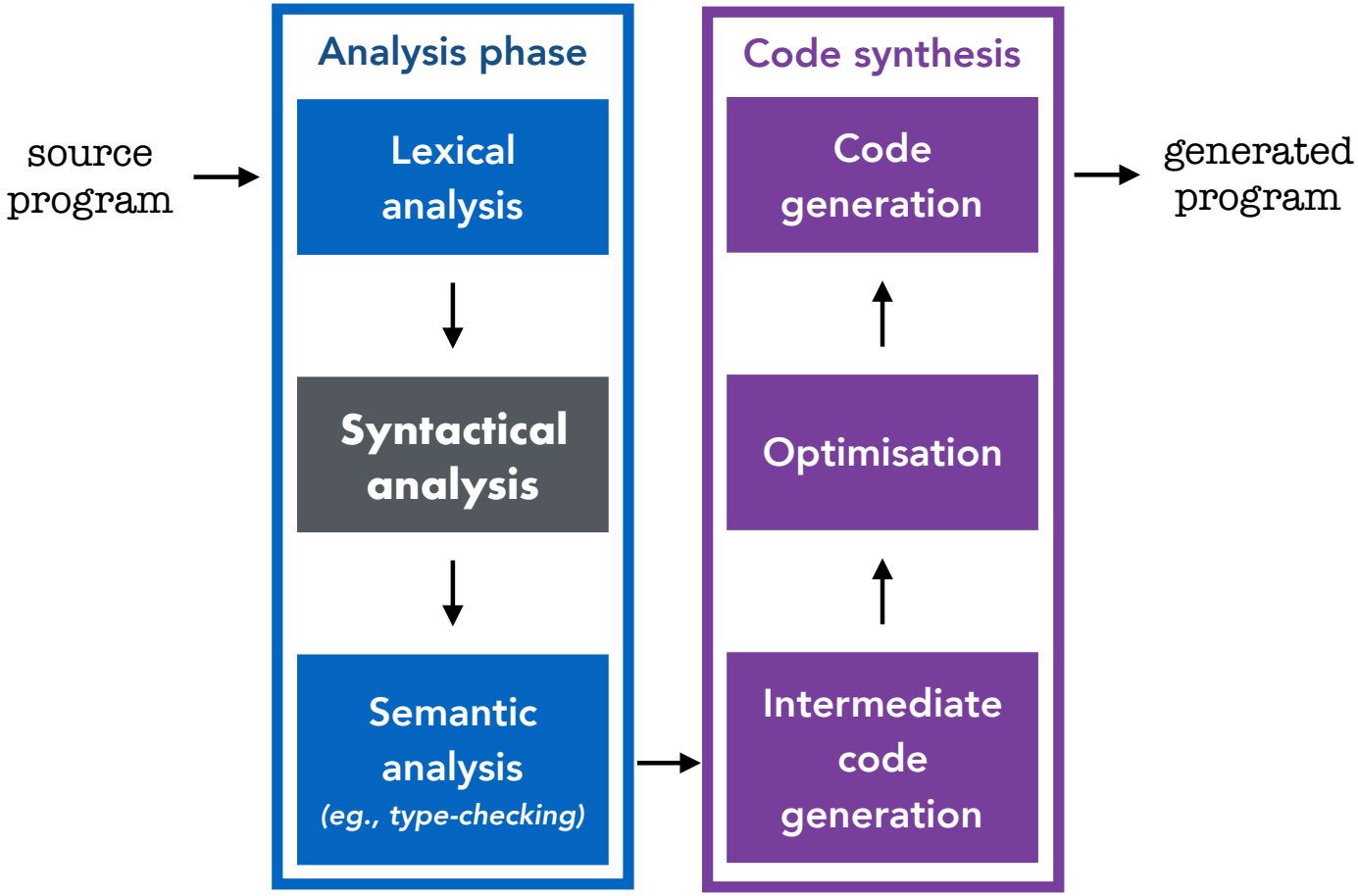
Compilers and computer architecture

7. Top-down parsing

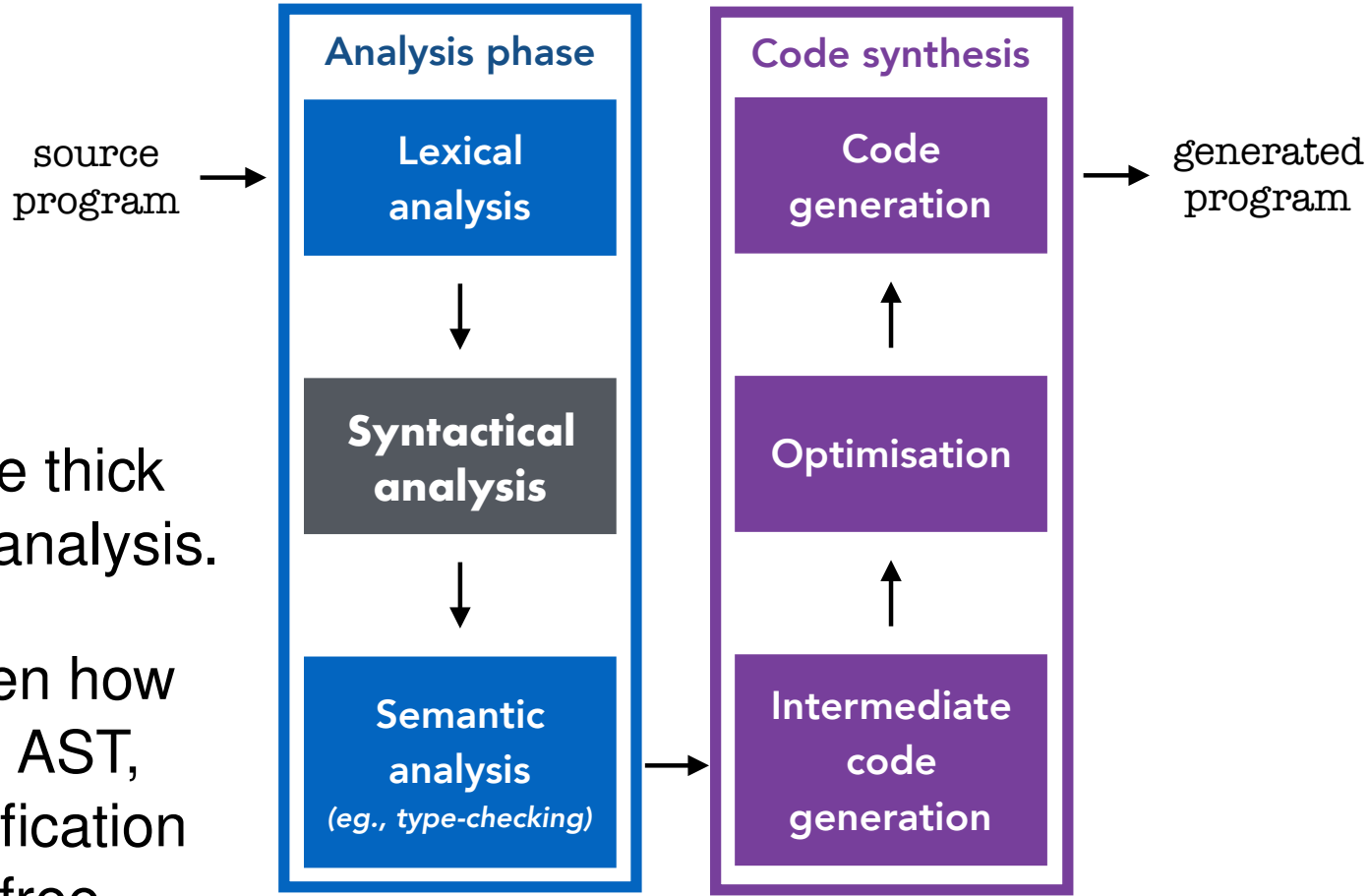
Ian Mackie

Autumn 2022

Where we are


source program
Analysis phase
Lexical analysis
Syntactical analysis
Semantic analysis
(eg., type-checking)
Code synthesis
Code generation
Optimisation
Intermediate code generation
generated program

Where we are



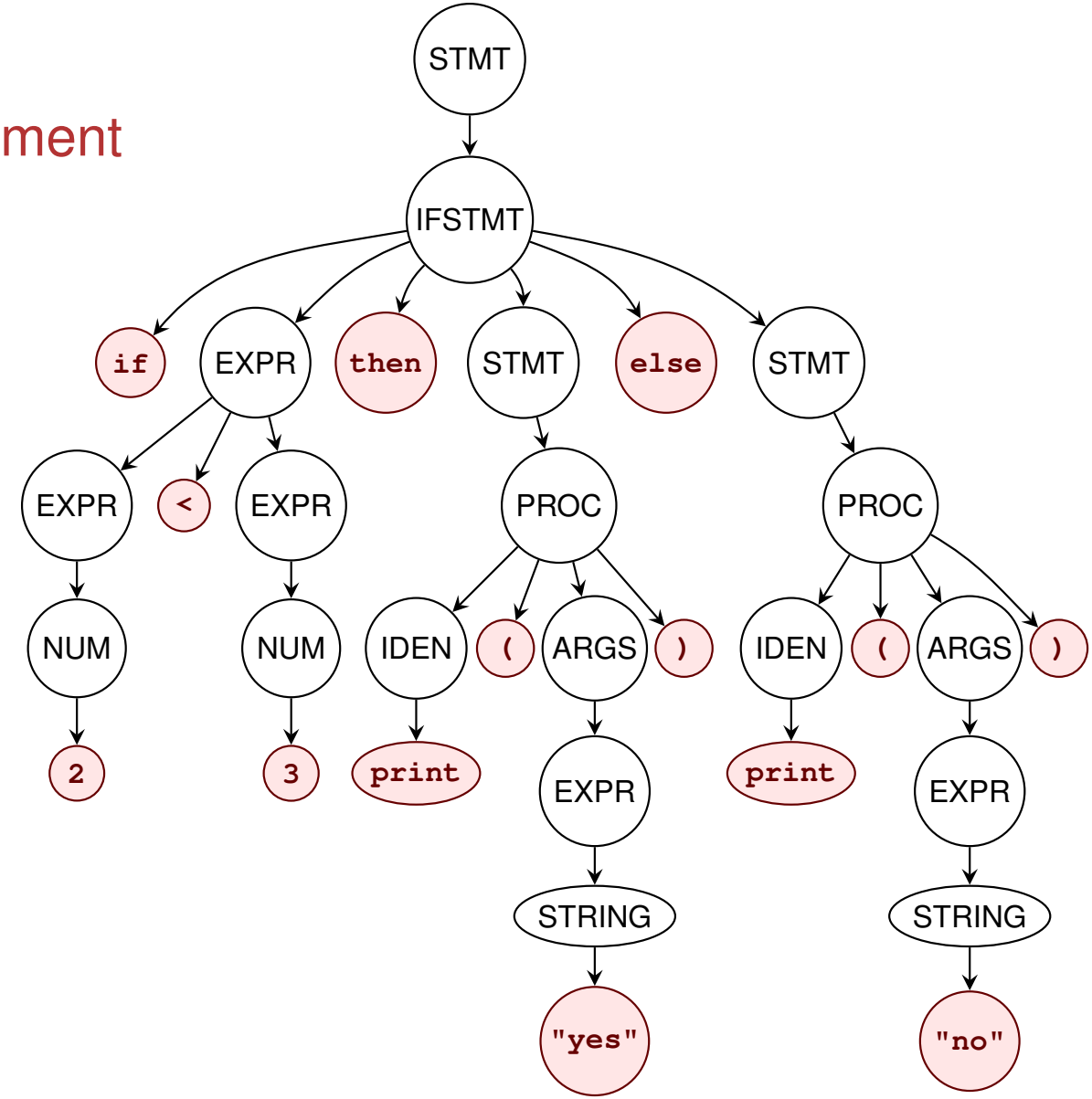
We are in the thick of syntactic analysis.

We have seen how to **define** an AST, from a specification of a context free grammar, and how it represents a *parse tree*.

But how do we **actually find** a parse tree for a source program?

Example

A parse tree for an if statement



Generated string:

```
if 2 < 3 then print ( "yes" ) else print ( "no" )
```

How do we find a parse tree?

That's all well and good —
but how could we actually **find** a derivation for an input string to build that parse tree?

How do we find a parse tree?

That's all well and good —
but how could we actually **find** a derivation for an input string
to build that parse tree?

This is what **parsing** is:

How do we find a parse tree?

That's all well and good —
but how could we actually **find** a derivation for an input string to build that parse tree?

This is what **parsing** is:

- finding such a derivation

How do we find a parse tree?

That's all well and good —
but how could we actually **find** a derivation for an input string to build that parse tree?

This is what **parsing** is:

- finding such a derivation
- and (usually), also building a parse tree / AST as it does so.

We will first focus just on finding the derivation, referring to parse trees but not worrying yet about building the AST.

How do we find a parse tree?

That's all well and good —
but how could we actually **find** a derivation for an input string to build that parse tree?

This is what **parsing** is:

- finding such a derivation
- and (usually), also building a parse tree / AST as it does so.

We will first focus just on finding the derivation, referring to parse trees but not worrying yet about building the AST.

Note — it's important that the grammar is unambiguous

(or that we have ways to select between different ways that an input may be parsed)

Two approaches to parsing

There are two high-level strategies for parsing:

- **Bottom up** (also known as shift-reduce algorithms)
- **Top down** (or predictive)

Two approaches to parsing

There are two high-level strategies for parsing:

- **Bottom up** (also known as shift-reduce algorithms)
  - Good: Fast, can deal with all (non-ambiguous) CFGs.
  - Bad: Complicated, hard to write by hand.
- **Top down** (or predictive)

Two approaches to parsing

There are two high-level strategies for parsing:

- **Bottom up** (also known as shift-reduce algorithms)
  - Good: Fast, can deal with all (non-ambiguous) CFGs.
  - Bad: Complicated, hard to write by hand.
- **Top down** (or predictive)
  - Good: conceptually easy, can be hand-written, powerful.

Two approaches to parsing

There are two high-level strategies for parsing:

- **Bottom up** (also known as shift-reduce algorithms)
  - Good: Fast, can deal with all (non-ambiguous) CFGs.
  - Bad: Complicated, hard to write by hand.
- **Top down** (or predictive)
  - Good: conceptually easy, can be hand-written, powerful.
  - Bad: Can be slow

Two approaches to parsing

There are two high-level strategies for parsing:

- **Bottom up** (also known as shift-reduce algorithms)
  - Good: Fast, can deal with all (non-ambiguous) CFGs.
  - Bad: Complicated, hard to write by hand.
- **Top down** (or predictive)
  - Good: conceptually easy, can be hand-written, powerful.
  - Bad: Can be slow

    Can't deal with left-recursive CFGs

    — but, left-recursive grammars can be converted automatically to non-left-recursive CFGs.

Two approaches to parsing

There are two high-level strategies for parsing:

- **Bottom up** (also known as shift-reduce algorithms)
  - Good: Fast, can deal with all (non-ambiguous) CFGs.
  - Bad: Complicated, hard to write by hand.
- **Top down** (or predictive)
  - Good: conceptually easy, can be hand-written, powerful.
  - Bad: Can be slow

    Can't deal with left-recursive CFGs

    — but, left-recursive grammars can be converted automatically to non-left-recursive CFGs.

In this and the next lecture, we consider top-down parsing.

Top down parsing
— intuition for the decision problem

Basic idea: *Trial substitution* —
try different rewrites, left-to-right, to achieve a derivation

We use a “partial derivation” (a string of terminals / nonterminals), and try different substitutions which might lead to a derivation.

Top down parsing
— intuition for the decision problem

Basic idea: *Trial substitution* —
try different rewrites, left-to-right, to achieve a derivation

We use a "partial derivation" (a string of terminals / nonterminals), and try different substitutions which might lead to a derivation.

- We start first with just the initial variable

Top down parsing
— intuition for the decision problem

Basic idea: *Trial substitution* —
try different rewrites, left-to-right, to achieve a derivation

We use a "partial derivation" (a string of terminals / nonterminals), and try different substitutions which might lead to a derivation.

- We start first with just the initial variable
- At each step, we refine the partial derivation by and examining the resulting string of symbols left-to-right:
  - taking any terminals produced, and matching them against input tokens (looking out for mismatches);
  - taking any non-terminals produced, and recursively testing different productions to try to match the rest of the input.

  This will eventually produce either derivation of the input, or allow us to or conclude that none exists.

Top down parsing
— intuition for the decision problem

Basic idea: *Trial substitution* —
try different rewrites, left-to-right, to achieve a derivation

We use a "partial derivation" (a string of terminals / nonterminals), and try different substitutions which might lead to a derivation.

- We start first with just the initial variable
- At each step, we refine the partial derivation by and examining the resulting string of symbols left-to-right:
  - taking any terminals produced, and matching them against input tokens (looking out for mismatches);
  - taking any non-terminals produced, and recursively testing different productions to try to match the rest of the input.

  This will eventually produce either derivation of the input, or allow us to or conclude that none exists.

A "depth-first search" for derivations of the input.

Top down parsing
— sketch of the algorithm

1. **For each possible reduction for the leftmost variable:** repeat the following until the reduction is either accepted or rejected.

   (‘Unmatch’ any tokens which are ‘matched’ in any rejected reduction.)

Top down parsing
— sketch of the algorithm

1. **For each possible reduction for the leftmost variable:** repeat the following until the reduction is either accepted or rejected.

   (‘Unmatch’ any tokens which are ‘matched’ in any rejected reduction.)

   (a) If all symbols in the partial derivation have been matched, match up with input tokens:

   - If all input tokens have been matched, **accept** — otherwise, reject this reduction and **backtrack**.

Top down parsing
— sketch of the algorithm

1. **For each possible reduction for the leftmost variable:** repeat the following until the reduction is either accepted or rejected.

   ('Unmatch' any tokens which are 'matched' in any rejected reduction.)

   (a) If all symbols in the partial derivation have been matched, match up with input tokens:
   - If all input tokens have been matched, **accept** — otherwise, reject this reduction and **backtrack**.

   (b) If the next unmatched symbols in the partial derivation is a **terminal**: compare it to the next unmatched input token.
   - If there **is** a 'next' input token, and they agree, **match** the two tokens and consider the next symbol in the partial derivation.
   - Otherwise: reject this reduction and **backtrack**.

Top down parsing
— sketch of the algorithm

1. **For each possible reduction for the leftmost variable:** repeat the following until the reduction is either accepted or rejected.

   ('Unmatch' any tokens which are 'matched' in any rejected reduction.)

   (a) If all symbols in the partial derivation have been matched, match up with input tokens:
   - If all input tokens have been matched, **accept** — otherwise, reject this reduction and **backtrack**.

   (b) If the next unmatched symbols in the partial derivation is a **terminal**: compare it to the next unmatched input token.
   - If there **is** a 'next' input token, and they agree, **match** the two tokens and consider the next symbol in the partial derivation.
   - Otherwise: reject this reduction and **backtrack**.

   (c) If the next unmatched terminal in the partial derivation is a **non-terminal**: recursively call this subroutine for this non-terminal.
   - If it accepts, then **accept** this reduction.
   - If it rejects: reject this reduction and **backtrack**.

Top down parsing
— sketch of the algorithm

1. **For each possible reduction for the leftmost variable:** repeat the following until the reduction is either accepted or rejected.

   ('Unmatch' any tokens which are 'matched' in any rejected reduction.)

   (a) If all symbols in the partial derivation have been matched, match up with input tokens:
   - If all input tokens have been matched, **accept** — otherwise, reject this reduction and **backtrack**.

   (b) If the next unmatched symbols in the partial derivation is a **terminal**: compare it to the next unmatched input token.
   - If there **is** a 'next' input token, and they agree, **match** the two tokens and consider the next symbol in the partial derivation.
   - Otherwise: reject this reduction and **backtrack**.

   (c) If the next unmatched terminal in the partial derivation is a **non-terminal**: recursively call this subroutine for this non-terminal.
   - If it accepts, then **accept** this reduction.
   - If it rejects: reject this reduction and **backtrack**.

2. 'Accept' if some reduction has been accepted; otherwise 'reject'.

Example top-down parsing

A grammar for a toy programming language:

Example top-down parsing

A grammar for a toy programming language:

$$
\begin{aligned}
\text{PROG} &\rightarrow \texttt{begin}\ \text{SUB} \\
&\mid \texttt{stmt} \quad \textit{// ... a stub for something more complex} \\
\text{SUB} &\rightarrow \text{PROG}\ \texttt{;}\ \text{SUB} \\
&\mid \texttt{end}
\end{aligned}
$$

Let ‘PROG’ be the initial variable.

Example top-down parsing

A grammar for a toy programming language:

$$
\begin{aligned}
\text{PROG} &\rightarrow \texttt{begin}\ \text{SUB} \\
&\mid \texttt{stmt} \quad \textit{// ... a stub for something more complex} \\
\text{SUB} &\rightarrow \text{PROG}\ \texttt{;}\ \text{SUB} \\
&\mid \texttt{end}
\end{aligned}
$$

Let ‘PROG’ be the initial variable.

We consider the following example string:

```
begin stmt; stmt; end
```

Example top-down parsing

A grammar for a toy programming language:

$$\begin{aligned} \text{PROG} &\to \texttt{begin}\ \text{SUB} \\ &\mid \texttt{stmt} \quad \textit{// ... a stub for something more complex} \\ \text{SUB} &\to \text{PROG}\ \texttt{;}\ \text{SUB} \\ &\mid \texttt{end} \end{aligned}$$

Let ‘PROG’ be the initial variable.

We consider the following example string:

```
begin stmt; stmt; end
```

Plan: starting from the initial variable —

Example top-down parsing

A grammar for a toy programming language:

$$\begin{aligned} \text{PROG} &\to \texttt{begin}\ \text{SUB} \\ &\mid \texttt{stmt} \quad \textit{// ... a stub for something more complex} \\ \text{SUB} &\to \text{PROG}\ \texttt{;}\ \text{SUB} \\ &\mid \texttt{end} \end{aligned}$$

Let ‘PROG’ be the initial variable.

We consider the following example string:

```
begin stmt; stmt; end
```

Plan: starting from the initial variable —

- Search for a rule which rewrites the variables to yield characters from the alphabet, **consistent with the input**.

Example top-down parsing

A grammar for a toy programming language:

$$\begin{aligned} \text{PROG} &\to \texttt{begin}\ \text{SUB} \\ &\mid \texttt{stmt} \quad \textit{// ... a stub for something more complex} \\ \text{SUB} &\to \text{PROG}\ \texttt{;}\ \text{SUB} \\ &\mid \texttt{end} \end{aligned}$$

Let ‘PROG’ be the initial variable.

We consider the following example string:

```
begin stmt; stmt; end
```

Plan: 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, note which rules you applied how — by constructing a parse tree

Example top-down parsing

We consider a process to match tokens to reduction rules by trial substitution:

Example top-down parsing

We consider a process to match tokens to reduction rules by trial substitution:

Tokens
`begin stmt ; stmt ; end`

Example top-down parsing

We consider a process to match tokens to reduction rules by trial substitution:

Matched	Next	Tokens
		`begin stmt ; stmt ; end`

Example top-down parsing

We consider a process to match tokens to reduction rules by trial substitution:

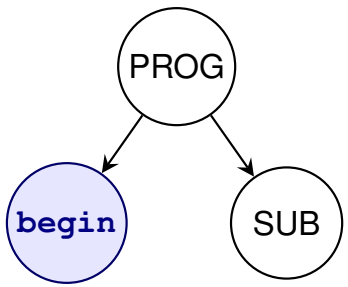


Matched	Next	Tokens
		`begin stmt ; stmt ; end`

Trial substitutions (leftmost-first):

Example top-down parsing

We consider a process to match tokens to reduction rules by trial substitution:



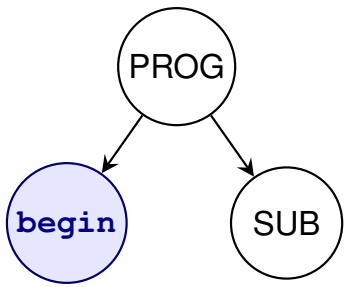
Matched	Next	Tokens
		`begin stmt ; stmt ; end`

Trial substitutions (leftmost-first):

- PROG $\rightarrow$ **`begin`** SUB

Example top-down parsing

We consider a process to match tokens to reduction rules by trial substitution:



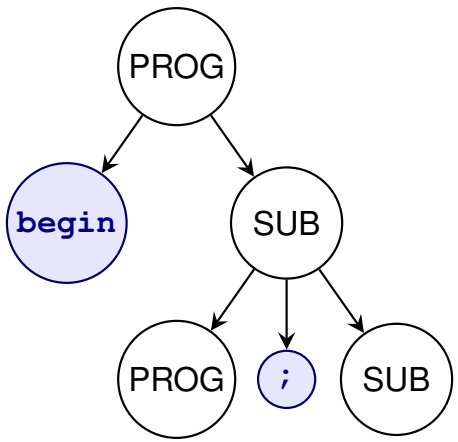
Matched	Next	Tokens
`begin`	`stmt ; stmt ; end`	

Trial substitutions (leftmost-first):

- PROG $\rightarrow$ **`begin`** SUB

Example top-down parsing

We consider a process to match tokens to reduction rules by trial substitution:



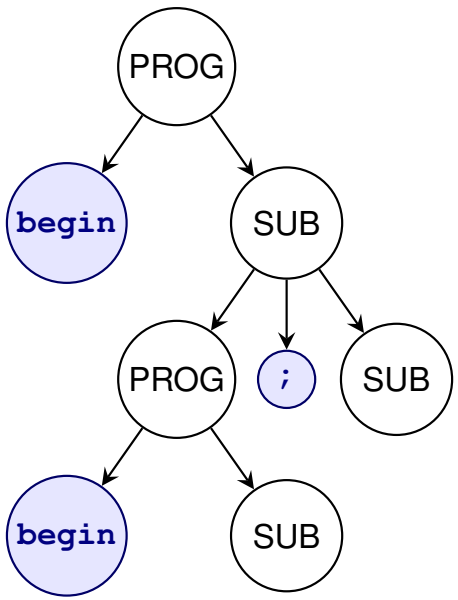
Matched	Next	Tokens
`begin`	`stmt ; stmt ; end`	

Trial substitutions (leftmost-first):

- PROG $\rightarrow$ `begin` SUB
- SUB $\rightarrow$ PROG `;` SUB

Example top-down parsing

We consider a process to match tokens to reduction rules by trial substitution:



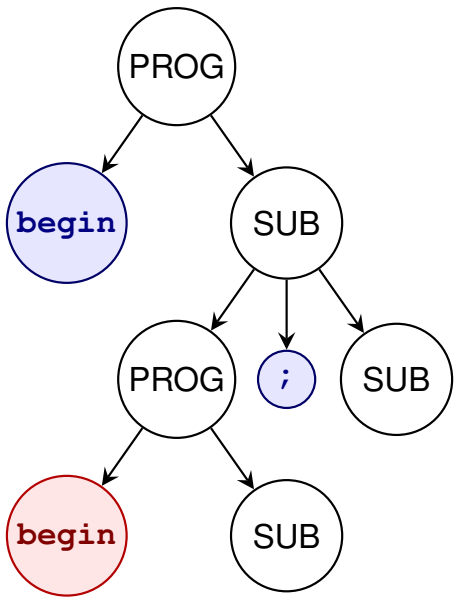
Matched	Next	Tokens
`begin`	`stmt ; stmt ; end`	

Trial substitutions (leftmost-first):

- PROG $\rightarrow$ `begin` SUB
- SUB $\rightarrow$ PROG `;` SUB
- PROG $\rightarrow$ `begin` SUB

Example top-down parsing

We consider a process to match tokens to reduction rules by trial substitution:



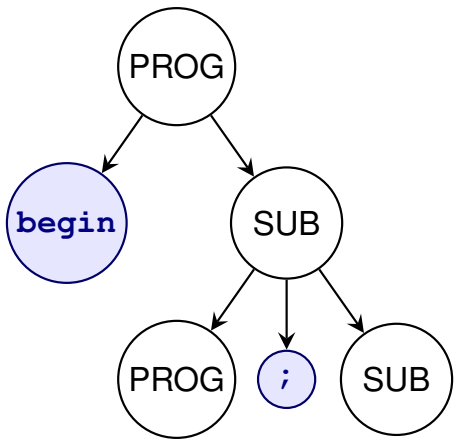
Matched	Next	Tokens
`begin`	`stmt ; stmt ; end`	

Trial substitutions (leftmost-first):

- PROG $\rightarrow$ `begin` SUB
- SUB $\rightarrow$ PROG `;` SUB
- ~~PROG $\rightarrow$ `begin` SUB~~

Example top-down parsing

We consider a process to match tokens to reduction rules by trial substitution:



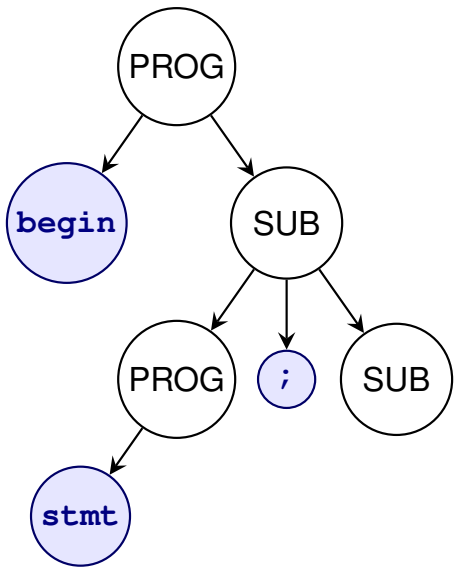
Matched	Next	Tokens
`begin`	`stmt ; stmt ; end`	

Trial substitutions (leftmost-first):

- PROG $\rightarrow$ `begin` SUB
- SUB $\rightarrow$ PROG `;` SUB

Example top-down parsing

We consider a process to match tokens to reduction rules by trial substitution:



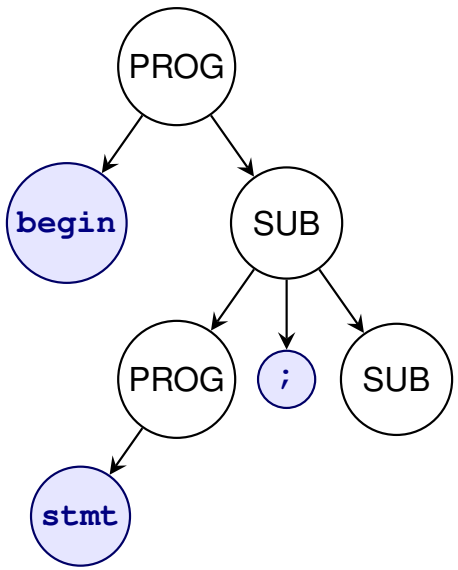
Matched	Next	Tokens
`begin`	`stmt ; stmt ; end`	

Trial substitutions (leftmost-first):

- PROG $\rightarrow$ `begin` SUB
- SUB $\rightarrow$ PROG `;` SUB
- PROG $\rightarrow$ `stmt`

Example top-down parsing

We consider a process to match tokens to reduction rules by trial substitution:



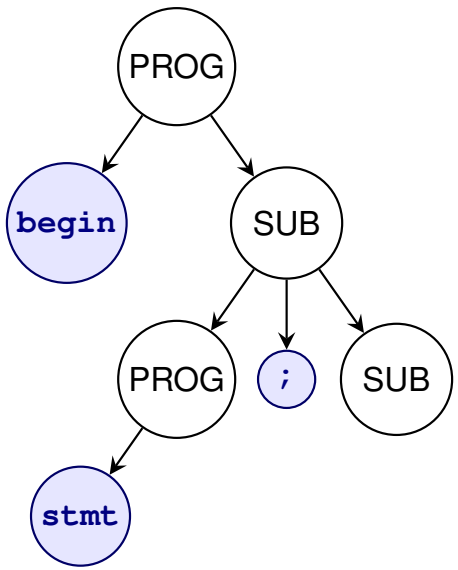
Matched	Next Tokens
`begin stmt`	`; stmt ; end`

Trial substitutions (leftmost-first):

- PROG $\rightarrow$ **`begin`** SUB
- SUB $\rightarrow$ PROG **`;`** SUB
- PROG $\rightarrow$ **`stmt`**

Example top-down parsing

We consider a process to match tokens to reduction rules by trial substitution:



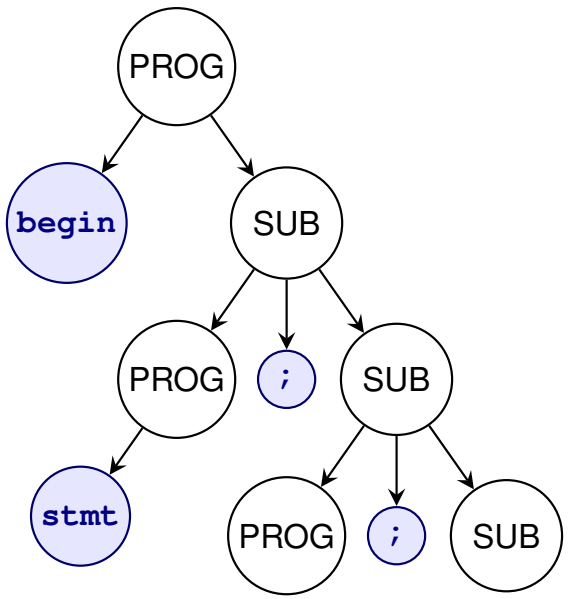
Matched	Next Tokens
`begin stmt ;`	`stmt ; end`

Trial substitutions (leftmost-first):

- PROG $\rightarrow$ **`begin`** SUB
- SUB $\rightarrow$ PROG **`;`** SUB
- PROG $\rightarrow$ **`stmt`**

Example top-down parsing

We consider a process to match tokens to reduction rules by trial substitution:



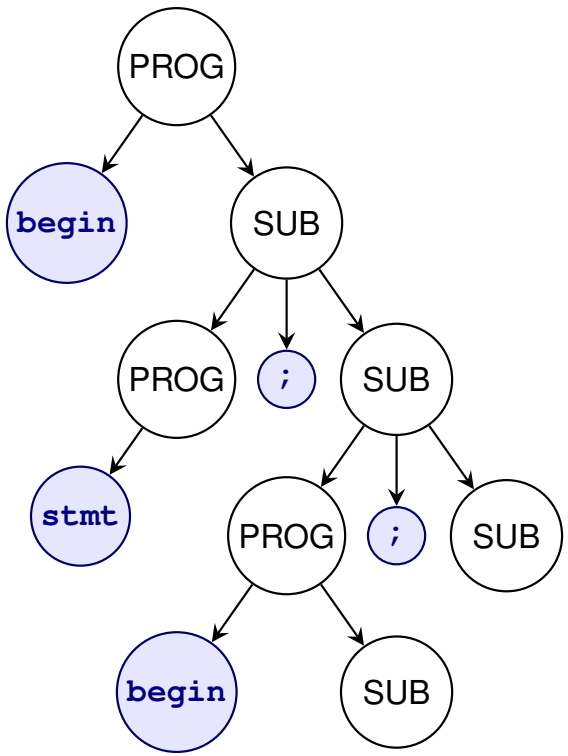
Matched	Next Tokens
`begin stmt ;`	`stmt ; end`

Trial substitutions (leftmost-first):

- PROG → **`begin`** SUB
- SUB → PROG **`;`** SUB
- PROG → **`stmt`**
- SUB → PROG **`;`** SUB

Example top-down parsing

We consider a process to match tokens to reduction rules by trial substitution:



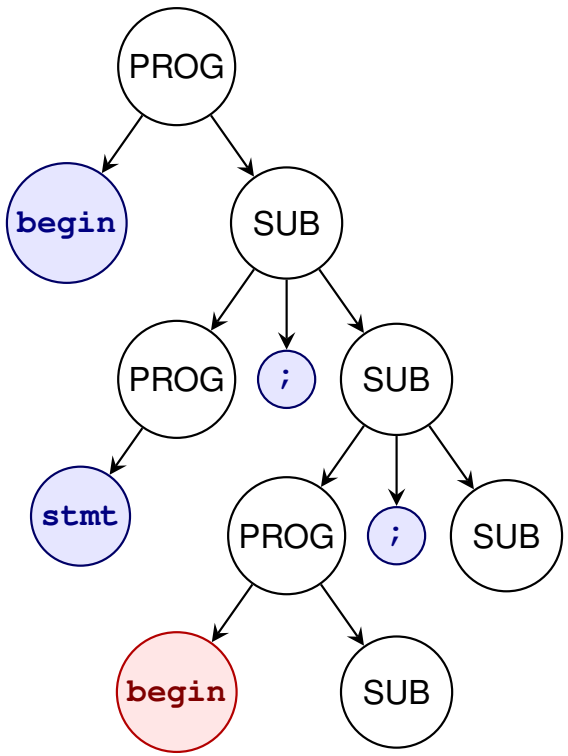
Matched	Next Tokens
`begin stmt ;`	`stmt ; end`

Trial substitutions (leftmost-first):

- PROG → **`begin`** SUB
- SUB → PROG **`;`** SUB
- PROG → **`stmt`**
- SUB → PROG **`;`** SUB
- PROG → **`begin`** SUB

Example top-down parsing

We consider a process to match tokens to reduction rules by trial substitution:



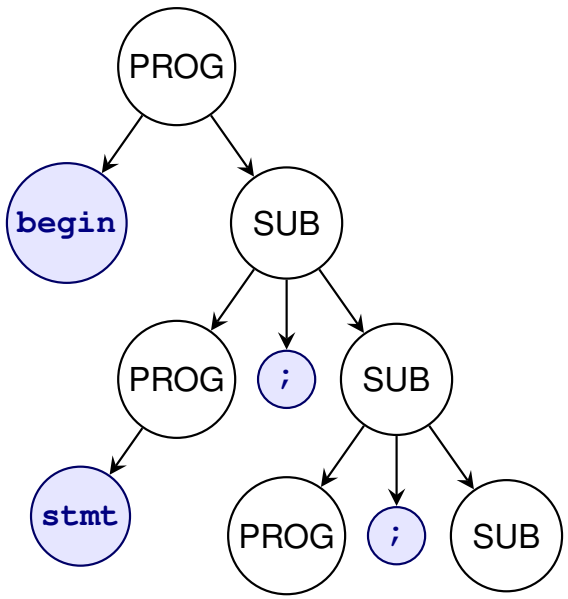
Matched	Next Tokens
`begin stmt ;`	`stmt ; end`

Trial substitutions (leftmost-first):

- PROG → **begin** SUB
- SUB → PROG ; SUB
- PROG → **stmt**
- SUB → PROG ; SUB
- ~~PROG → **begin** SUB~~

Example top-down parsing

We consider a process to match tokens to reduction rules by trial substitution:



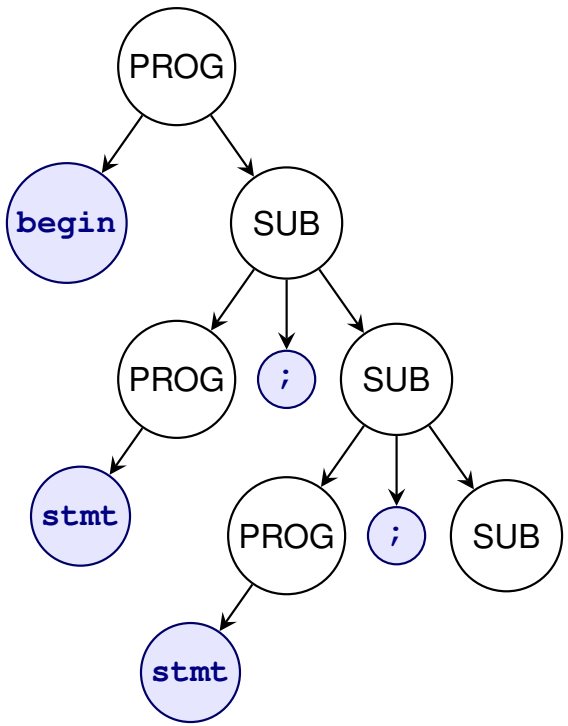
Matched	Next Tokens
`begin stmt ;`	`stmt ; end`

Trial substitutions (leftmost-first):

- PROG → **`begin`** SUB
- SUB → PROG **`;`** SUB
- PROG → **`stmt`**
- SUB → PROG **`;`** SUB

Example top-down parsing

We consider a process to match tokens to reduction rules by trial substitution:



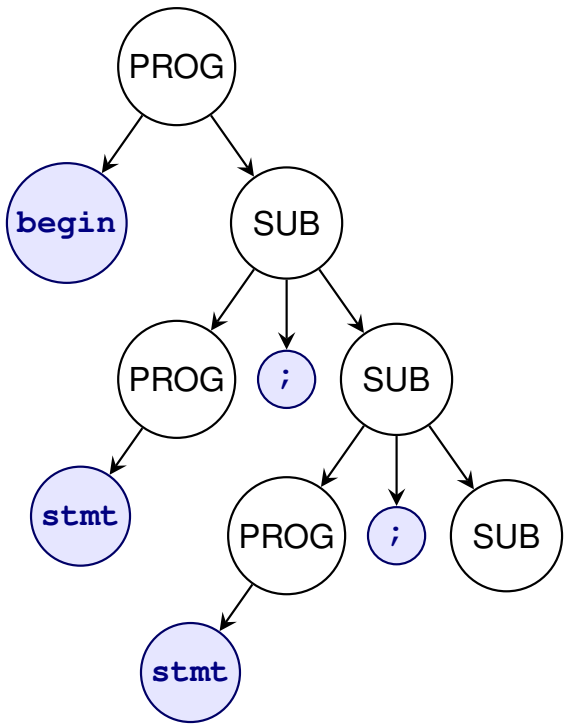
Matched	Next Tokens
`begin stmt ;`	`stmt ; end`

Trial substitutions (leftmost-first):

- PROG → **`begin`** SUB
- SUB → PROG **`;`** SUB
- PROG → **`stmt`**
- SUB → PROG **`;`** SUB
- PROG → **`stmt`**

Example top-down parsing

We consider a process to match tokens to reduction rules by trial substitution:



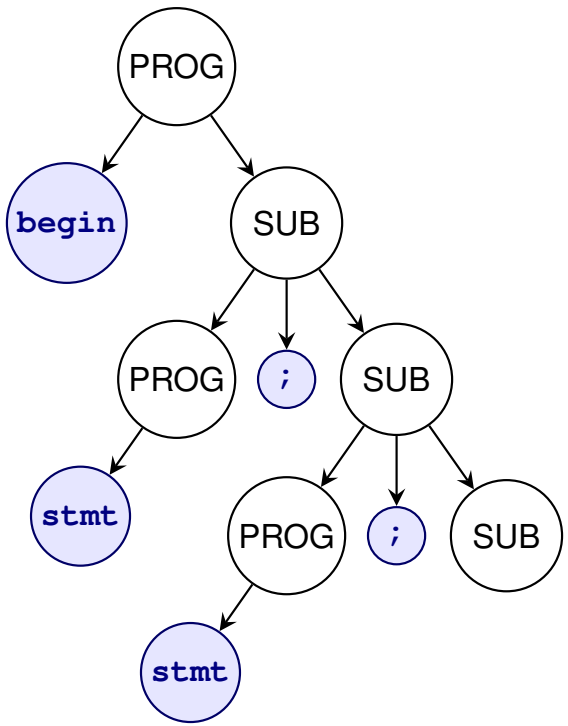
Matched	Next Tokens
`begin stmt ; stmt`	`;` `end`

Trial substitutions (leftmost-first):

- PROG → **`begin`** SUB
- SUB → PROG **`;`** SUB
- PROG → **`stmt`**
- SUB → PROG **`;`** SUB
- PROG → **`stmt`**

Example top-down parsing

We consider a process to match tokens to reduction rules by trial substitution:



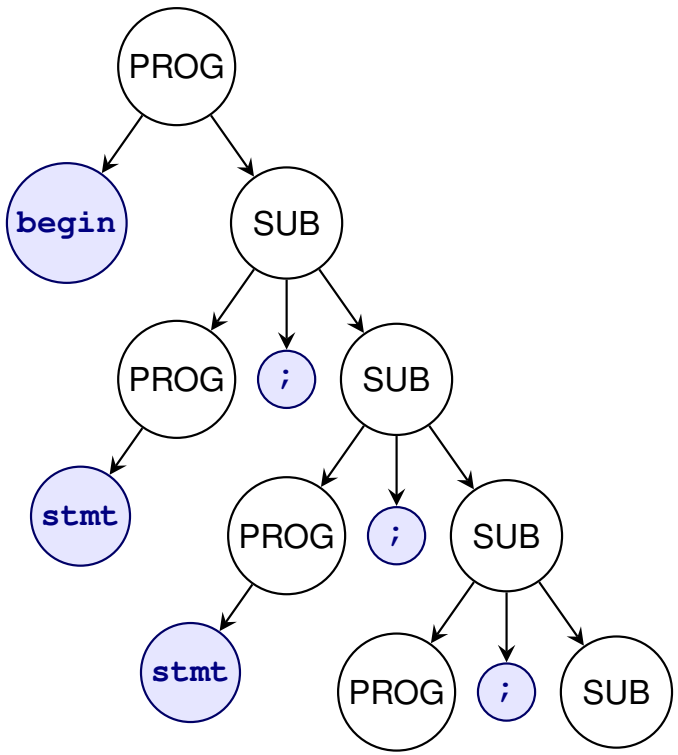
Matched	Next Tokens
`begin stmt ; stmt ;`	`end`

Trial substitutions (leftmost-first):

- PROG → **`begin`** SUB
- SUB → PROG **`;`** SUB
- PROG → **`stmt`**
- SUB → PROG **`;`** SUB
- PROG → **`stmt`**

Example top-down parsing

We consider a process to match tokens to reduction rules by trial substitution:



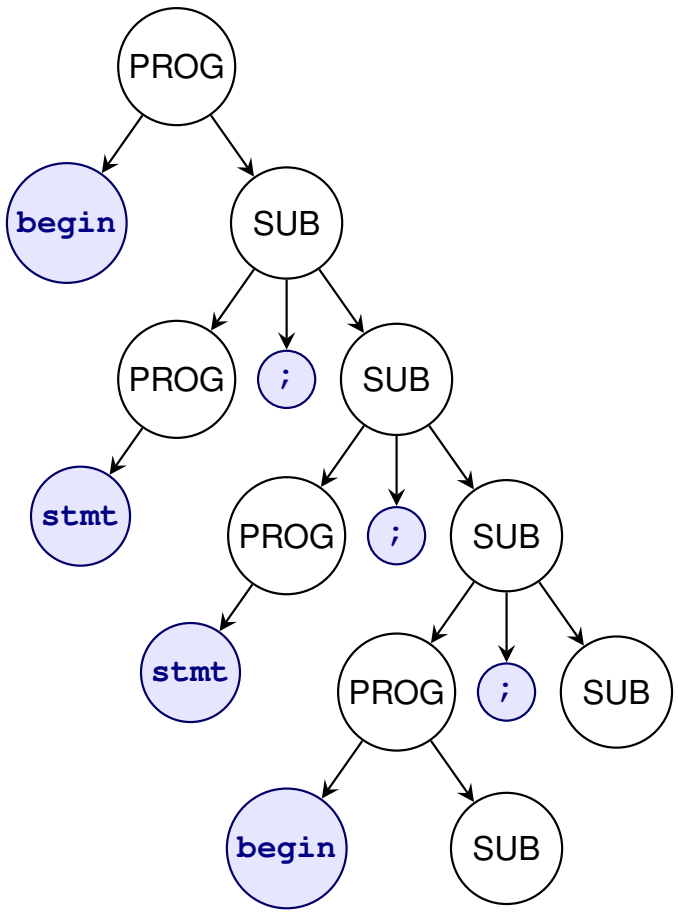
Matched	Next Tokens
`begin stmt ; stmt ;`	`end`

Trial substitutions (leftmost-first):

- PROG → **`begin`** SUB
- SUB → PROG **`;`** SUB
- PROG → **`stmt`**
- SUB → PROG **`;`** SUB
- PROG → **`stmt`**
- SUB → PROG **`;`** SUB

Example top-down parsing

We consider a process to match tokens to reduction rules by trial substitution:



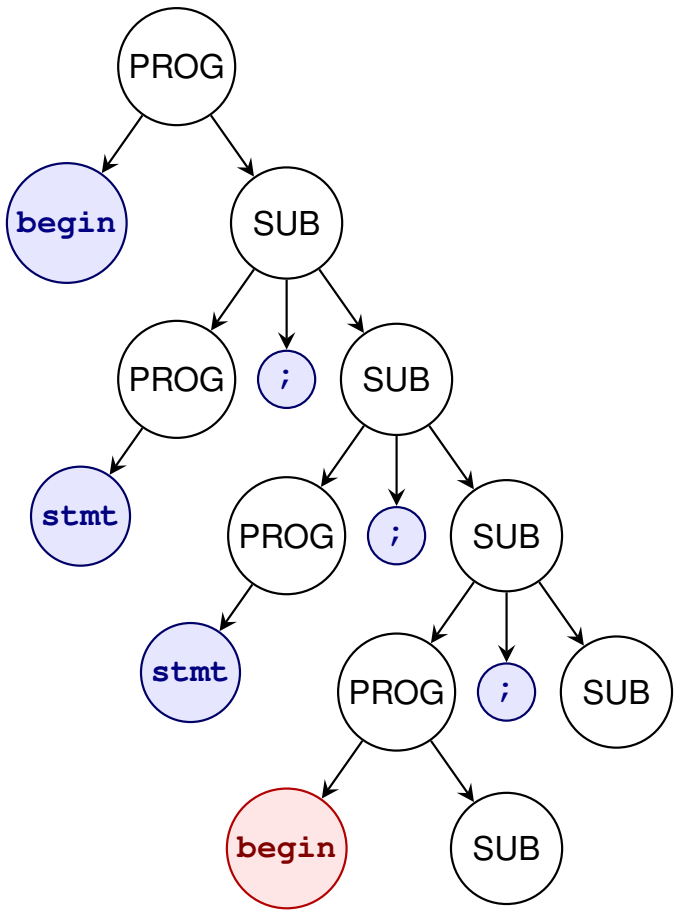
Matched	Next Tokens
`begin stmt ; stmt ;`	`end`

Trial substitutions (leftmost-first):

- PROG → **`begin`** SUB
- SUB → PROG **`;`** SUB
- PROG → **`stmt`**
- SUB → PROG **`;`** SUB
- PROG → **`stmt`**
- SUB → PROG **`;`** SUB
- PROG → **`begin`** SUB

Example top-down parsing

We consider a process to match tokens to reduction rules by trial substitution:



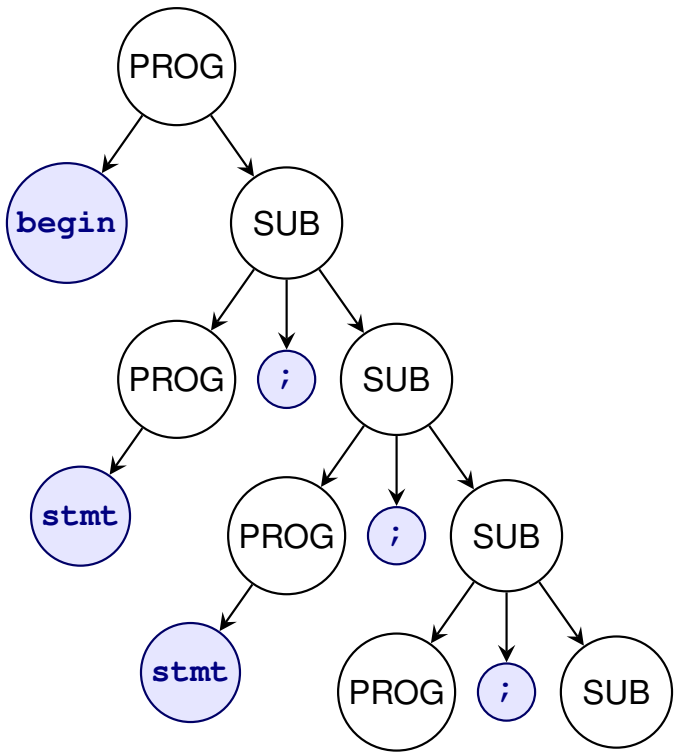
Matched	Next Tokens
`begin stmt ; stmt ;`	`end`

Trial substitutions (leftmost-first):

- PROG → **begin** SUB
- SUB → PROG ; SUB
- PROG → **stmt**
- SUB → PROG ; SUB
- PROG → **stmt**
- SUB → PROG ; SUB
- ~~PROG → **begin** SUB~~

Example top-down parsing

We consider a process to match tokens to reduction rules by trial substitution:



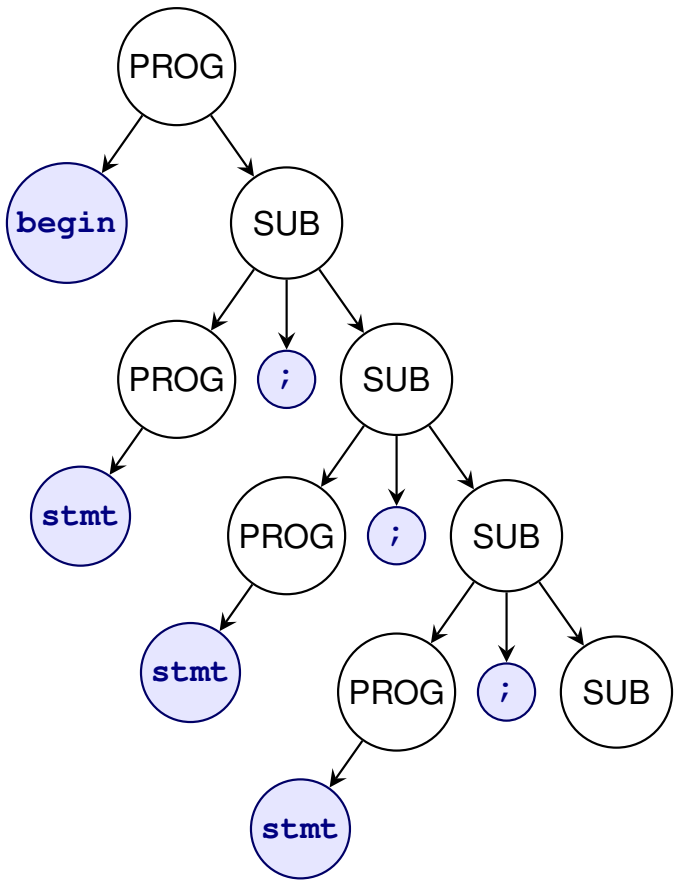
Matched	Next Tokens
`begin stmt ; stmt ;`	`end`

Trial substitutions (leftmost-first):

- PROG → **`begin`** SUB
- SUB → PROG **`;`** SUB
- PROG → **`stmt`**
- SUB → PROG **`;`** SUB
- PROG → **`stmt`**
- SUB → PROG **`;`** SUB

Example top-down parsing

We consider a process to match tokens to reduction rules by trial substitution:



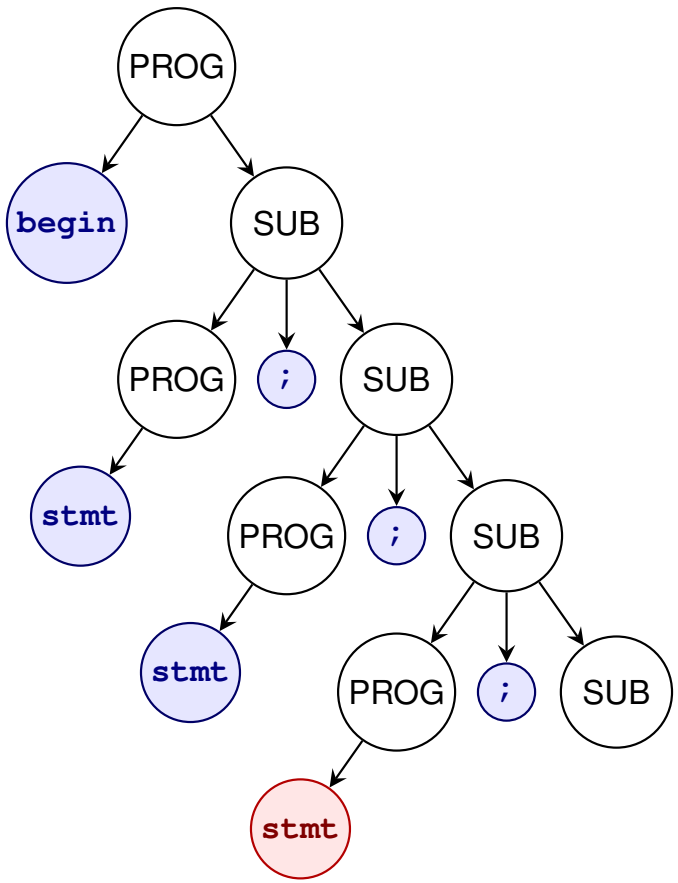
Matched	Next Tokens
`begin stmt ; stmt ;`	`end`

Trial substitutions (leftmost-first):

- PROG → **`begin`** SUB
- SUB → PROG **`;`** SUB
- PROG → **`stmt`**
- SUB → PROG **`;`** SUB
- PROG → **`stmt`**
- SUB → PROG **`;`** SUB
- PROG → **`stmt`**

Example top-down parsing

We consider a process to match tokens to reduction rules by trial substitution:



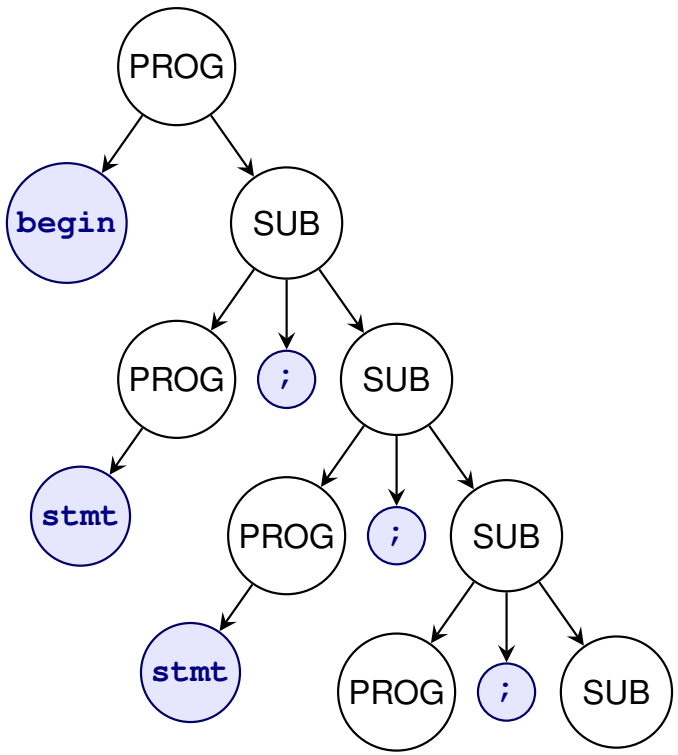
Matched	Next Tokens
`begin stmt ; stmt ;`	`end`

Trial substitutions (leftmost-first):

- PROG → **`begin`** SUB
- SUB → PROG **`;`** SUB
- PROG → **`stmt`**
- SUB → PROG **`;`** SUB
- PROG → **`stmt`**
- SUB → PROG **`;`** SUB
- ~~PROG → **`stmt`**~~

Example top-down parsing

We consider a process to match tokens to reduction rules by trial substitution:



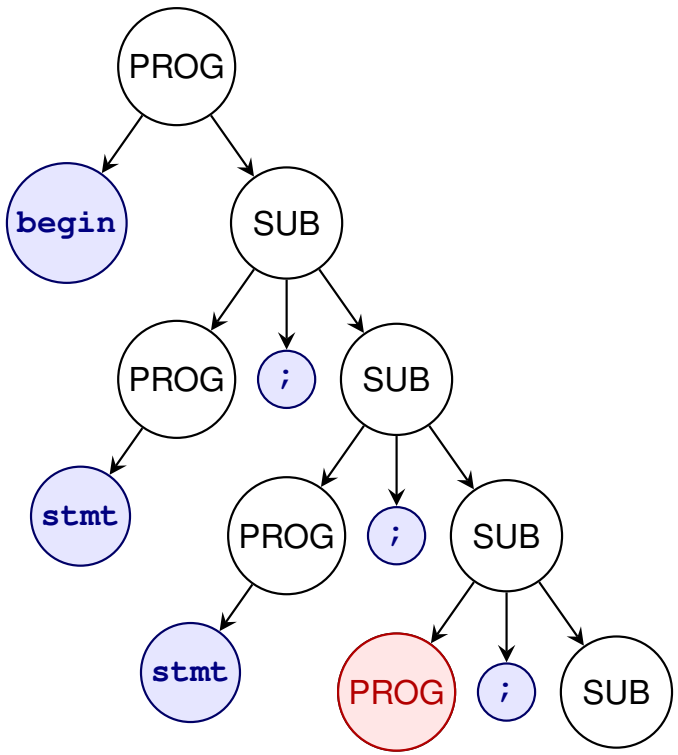
Matched	Next Tokens
`begin stmt ; stmt ;`	`end`

Trial substitutions (leftmost-first):

- PROG → **`begin`** SUB
- SUB → PROG **`;`** SUB
- PROG → **`stmt`**
- SUB → PROG **`;`** SUB
- PROG → **`stmt`**
- SUB → PROG **`;`** SUB

Example top-down parsing

We consider a process to match tokens to reduction rules by trial substitution:



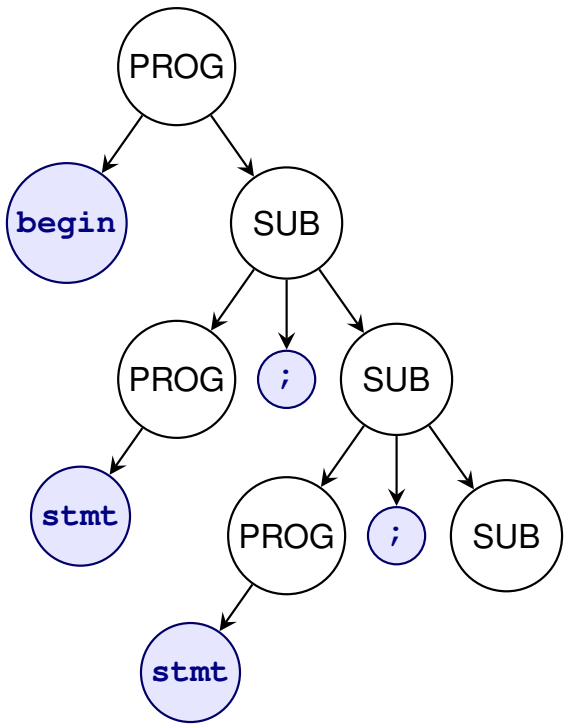
Matched	Next Tokens
`begin stmt ; stmt ;`	`end`

Trial substitutions (leftmost-first):

- PROG → **`begin`** SUB
- SUB → PROG **`;`** SUB
- PROG → **`stmt`**
- SUB → PROG **`;`** SUB
- PROG → **`stmt`**
- ~~SUB → PROG ; SUB~~

Example top-down parsing

We consider a process to match tokens to reduction rules by trial substitution:



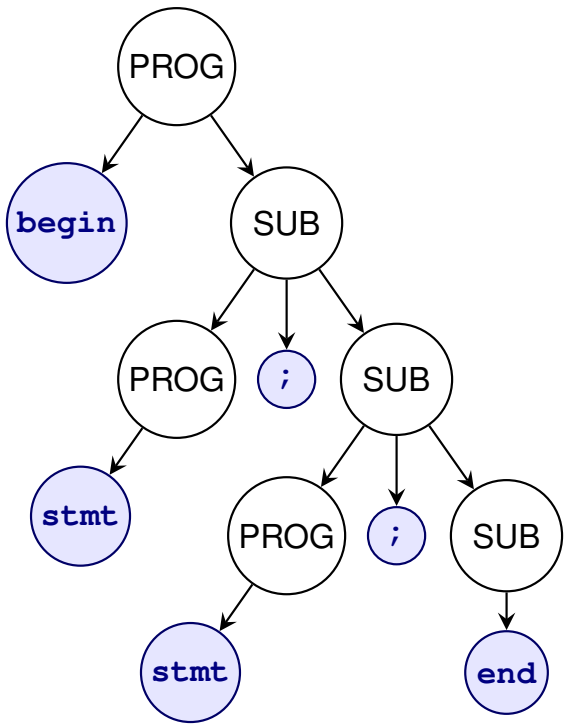
Matched	Next Tokens
`begin stmt ; stmt ;`	`end`

Trial substitutions (leftmost-first):

- PROG → **`begin`** SUB
- SUB → PROG **`;`** SUB
- PROG → **`stmt`**
- SUB → PROG **`;`** SUB
- PROG → **`stmt`**

Example top-down parsing

We consider a process to match tokens to reduction rules by trial substitution:



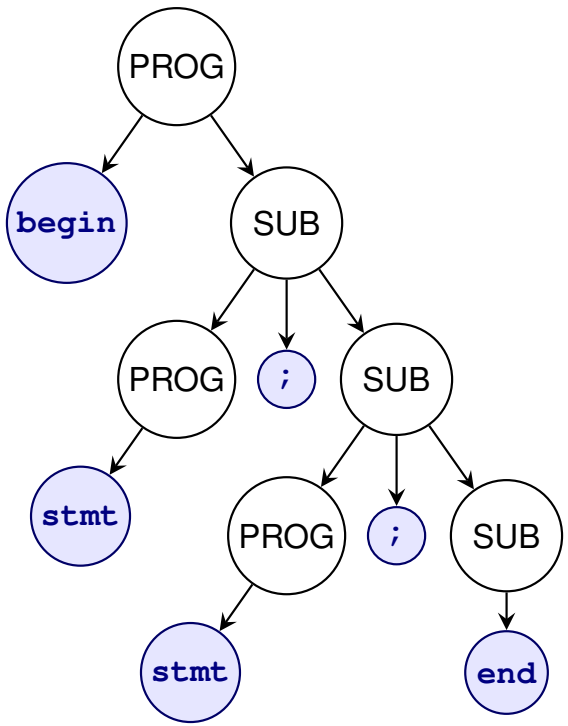
Matched	Next Tokens
`begin stmt ; stmt ;`	`end`

Trial substitutions (leftmost-first):

- PROG → **`begin`** SUB
- SUB → PROG **`;`** SUB
- PROG → **`stmt`**
- SUB → PROG **`;`** SUB
- PROG → **`stmt`**
- SUB → **`end`**

Example top-down parsing

We consider a process to match tokens to reduction rules by trial substitution:



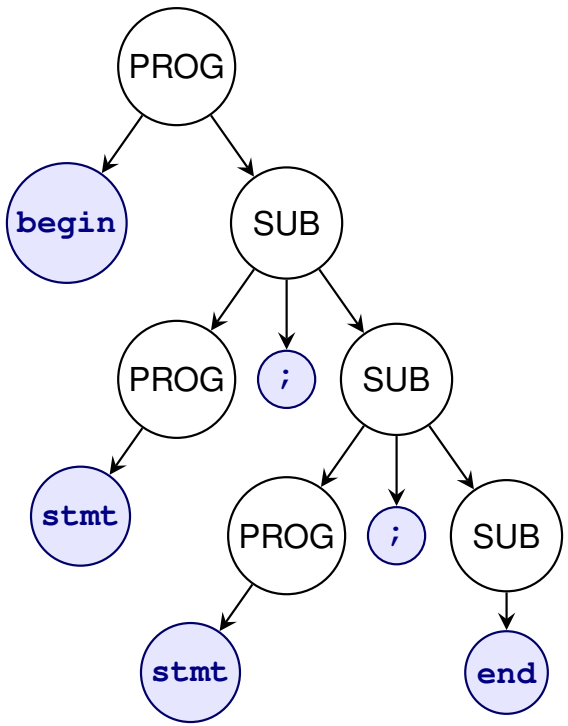
Matched	Next Tokens
`begin stmt ; stmt ; end`	

Trial substitutions (leftmost-first):

- PROG → **`begin`** SUB
- SUB → PROG **`;`** SUB
- PROG → **`stmt`**
- SUB → PROG **`;`** SUB
- PROG → **`stmt`**
- SUB → **`end`**

Example top-down parsing

We consider a process to match tokens to reduction rules by trial substitution:



Matched	Next Tokens
`begin stmt ; stmt ; end`	

Trial substitutions (leftmost-first):

- PROG → **`begin`** SUB
- SUB → PROG **`;`** SUB
- PROG → **`stmt`**
- SUB → PROG **`;`** SUB
- PROG → **`stmt`**
- SUB → **`end`**

In this way, we find a derivation for the program (specifically, a leftmost-derivation), matching tokens one at a time.

Summary of approach we used

1. Start with all input tokens unmatched, and a “partial derivation” consisting of just PROG.

Summary of approach we used

1. Start with all input tokens unmatched, and a “partial derivation” consisting of just PROG.
2. We recursively **attempt** / **consider**, productions for the leftmost variable which might lead to a match for all input tokens.
    - We record the productions which we apply, as we try them.

Summary of approach we used

1. Start with all input tokens unmatched, and a “partial derivation” consisting of just PROG.
2. We recursively **attempt** / **consider**, productions for the leftmost variable which might lead to a match for all input tokens.
    - We record the productions which we apply, as we try them.
    - Any reduction which leads to a match for some of the input tokens — or which introduces new variables before any new tokens — is investigated further.

Summary of approach we used

1. Start with all input tokens unmatched, and a “partial derivation” consisting of just PROG.
2. We recursively **attempt** / **consider**, productions for the leftmost variable which might lead to a match for all input tokens.
    - We record the productions which we apply, as we try them.
    - Any reduction which leads to a match for some of the input tokens — or which introduces new variables before any new tokens — is investigated further.
    - Any reduction which leads to a mismatch of tokens gets rejected — and all matches of tokens done in exploring the reduction are undone.

Summary of approach we used

1. Start with all input tokens unmatched, and a “partial derivation” consisting of just PROG.
2. We recursively **attempt** / **consider**, productions for the leftmost variable which might lead to a match for all input tokens.
   - We record the productions which we apply, as we try them.
   - Any reduction which leads to a match for some of the input tokens — or which introduces new variables before any new tokens — is investigated further.
   - Any reduction which leads to a mismatch of tokens gets rejected — and all matches of tokens done in exploring the reduction are undone.
3. If we manage to match all of the input tokens, with no unmatched symbols in the partial derivation, then we’ve succeeded in parsing the input.

Summary of approach we used

1. Start with all input tokens unmatched, and a “partial derivation” consisting of just PROG.
2. We recursively **attempt** / **consider**, productions for the leftmost variable which might lead to a match for all input tokens.
   - We record the productions which we apply, as we try them.
   - Any reduction which leads to a match for some of the input tokens — or which introduces new variables before any new tokens — is investigated further.
   - Any reduction which leads to a mismatch of tokens gets rejected — and all matches of tokens done in exploring the reduction are undone.
3. If we manage to match all of the input tokens, with no unmatched symbols in the partial derivation, then we’ve succeeded in parsing the input.
   - The recorded productions describe a parse tree for the original list of input tokens.

Summary of approach we used

1. Start with all input tokens unmatched, and a “partial derivation” consisting of just PROG.
2. We recursively **attempt** / **consider**, productions for the leftmost variable which might lead to a match for all input tokens.
   - We record the productions which we apply, as we try them.
   - Any reduction which leads to a match for some of the input tokens — or which introduces new variables before any new tokens — is investigated further.
   - Any reduction which leads to a mismatch of tokens gets rejected — and all matches of tokens done in exploring the reduction are undone.
3. If we manage to match all of the input tokens, with no unmatched symbols in the partial derivation, then we’ve succeeded in parsing the input.
   - The recorded productions describe a parse tree for the original list of input tokens.

Let’s turn this into pseudo-code!

Some code for top-down parsing

Setting up lists of mixed tokens and nonterminals

Recall our grammar.

$$
\begin{aligned}
\text{PROG} &\rightarrow \texttt{begin}\ \text{SUB} \\
&\mid\ \texttt{stmt} \quad \textit{// ... a stub for something more complex} \\
\text{SUB} &\rightarrow \text{PROG}\ \texttt{;}\ \text{SUB} \\
&\mid\ \texttt{end}
\end{aligned}
$$

Some code for top-down parsing

Setting up lists of mixed tokens and nonterminals

Recall our grammar.

$$\begin{aligned} \text{PROG} &\rightarrow \texttt{begin}\ \text{SUB} \\ &\mid\ \texttt{stmt} \quad \textit{// ... a stub for something more complex} \\ \text{SUB} &\rightarrow \text{PROG}\ \texttt{;}\ \text{SUB} \\ &\mid\ \texttt{end} \end{aligned}$$

First we define ‘abstract’ classes for tokens/terminals, for nonterminals, and for symbols which could be either

Some code for top-down parsing

Setting up lists of mixed tokens and nonterminals

Recall our grammar.

$$\begin{aligned} \text{PROG} &\rightarrow \texttt{begin}\ \text{SUB} \\ &\mid \texttt{stmt} \quad \textit{// ... a stub for something more complex} \\ \text{SUB} &\rightarrow \text{PROG}\ \texttt{;}\ \text{SUB} \\ &\mid \texttt{end} \end{aligned}$$

First we define 'abstract' classes for tokens/terminals, for nonterminals, and for symbols which could be either

```
class Symb {
  Symb () {}
  bool isToken () { return false; }
  String type ()  { return ""; } }

  bool isSameType (Symb s) {
    return (! type().equals("") &&
              type().equals(s.type()); } }
```

Some code for top-down parsing

Defining tokens

$$
\begin{aligned}
\text{PROG} &\rightarrow \texttt{begin}\ \text{SUB} \\
&\mid\ \texttt{stmt} \quad \textit{// ... a stub for something more complex} \\
\text{SUB} &\rightarrow \text{PROG}\ \texttt{;}\ \text{SUB} \\
&\mid\ \texttt{end}
\end{aligned}
$$

Some code for top-down parsing

Defining tokens

$$
\begin{aligned}
\text{PROG} &\rightarrow \texttt{begin}\ \text{SUB} \\
&\mid \texttt{stmt} \quad \textit{// ... a stub for something more complex} \\
\text{SUB} &\rightarrow \text{PROG}\ \texttt{;}\ \text{SUB} \\
&\mid \texttt{end}
\end{aligned}
$$

We define a class for each token type.

- In a more complex language, these might also store data such as an integer or an identifier name.

Some code for top-down parsing

Defining tokens

$$\begin{aligned} \text{PROG} &\rightarrow \texttt{begin}\ \text{SUB} \\ &\mid \texttt{stmt} \quad \textit{// ... a stub for something more complex} \\ \text{SUB} &\rightarrow \text{PROG}\ \texttt{;}\ \text{SUB} \\ &\mid \texttt{end} \end{aligned}$$

We define a class for each token type.

- In a more complex language, these might also store data such as an integer or an identifier name.

```
class Token extends Symb {
  Token () {}
  bool isToken () { return true; } }
class Begin extends Token {
  String type ()  { return "begin"; } }
class Stmt extends Token {
  String type ()  { return "stmt"; } }
class Semic extends Token {
  String type ()  { return ";" ; } }
class End extends Token {
  String type ()  { return "end"; } }
```

Some code for top-down parsing

Defining nonterminals

$$
\begin{aligned}
\text{PROG} &\rightarrow \texttt{begin}\ \text{SUB} \\
&\mid \texttt{stmt} \quad \textit{// ... a stub for something more complex} \\
\text{SUB} &\rightarrow \text{PROG}\ \texttt{;}\ \text{SUB} \\
&\mid \texttt{end}
\end{aligned}
$$

Some code for top-down parsing

Defining nonterminals

$$
\begin{aligned}
\text{PROG} &\rightarrow \texttt{begin}\ \text{SUB} \\
&\mid \texttt{stmt} \quad \textit{// ... a stub for something more complex} \\
\text{SUB} &\rightarrow \text{PROG}\ \texttt{;}\ \text{SUB} \\
&\mid \texttt{end}
\end{aligned}
$$

We define a class for each nonterminal.

- We provide ‘stub’ constructors which produce dummy objects
- The production “PROG $\rightarrow$ `begin` SUB” is basically a wrapper for “SUB”... so we also define “SUB” as an extension of “PROG”

Some code for top-down parsing

Defining nonterminals

PROG → **begin** SUB
| **stmt** *// ... a stub for something more complex*

SUB → PROG **;** SUB
| **end**

We define a class for each nonterminal.

- We provide 'stub' constructors which produce dummy objects
- The production "PROG → **begin** SUB" is basically a wrapper for "SUB"... so we also define "SUB" as an extension of "PROG"

```
class PROG extends Symb {
  PROG () {}
  String type() { return "PROG"; }
  PROG compact() { return this; } } // to be explained

class SUB extends PROG {
  SUB () {}
  String type() { return "SUB"; }
  SUB compact() { return this; } } // to be explained
```

Some code for top-down parsing

Indirection to try out different productions

What the `compact()` methods are for:

(This is particular to the approach we take, in this lecture, in Java)

- We define classes which which provide a level of indirection:

```
class TrialPROG extends PROG {
  PROG production;
  TrialPROG () { production = new PROG(); }
  PROG compact() { return production.compact(); } }

class TrialSUB extends SUB {
  SUB production;
  TrialSUB () { production = new SUB(); }
  SUB compact() { return production.compact(); } }
```

- This will allow us to try **different productions** for PROG, within a single object of type PROG (and similarly for SUB)
- `compact()` will help us to remove the extra level of indirection, after we're done

Some code for top-down parsing

How to do trial productions

$$\text{PROG} \rightarrow \texttt{begin}\ \text{SUB}$$

Some code for top-down parsing

How to do trial productions

PROG $\rightarrow$ **`begin`** SUB

A procedure to test this production at the beginning of a symbol list:

```
bool tryPROGproductionSUB(TrialPROG p, List<Symb>  symbs,
                                       List<Token> toks) {
  List<Symb> trialToks = new LinkedList<>(toks);
  List<Symb> trialSymbs = new LinkedList<>(symbs);
  Token beginTok = new Begin ();
  Symb  SUBsymb = new TrialSUB ();
  List<Symb> production = new LinkedList<>(
    Arrays.asList(beginTok, SUBsymb)));

  trialSymbs.addAll(0, production);
  p.production = SUBsymb;

  return parse(trialSymbs, trialToks); }
```

Some code for top-down parsing

How to do trial productions

PROG $\rightarrow$ **`begin`** SUB

A procedure to test this production at the beginning of a symbol list:

```
bool tryPROGproductionSUB(TrialPROG p, List<Symb>  symbs,
                                       List<Token> toks) {
  List<Symb> trialToks = new LinkedList<>(toks);
  List<Symb> trialSymbs = new LinkedList<>(symbs);
  Token beginTok = new Begin ();
  Symb  SUBsymb = new TrialSUB ();
  List<Symb> production = new LinkedList<>(
    Arrays.asList(beginTok, SUBsymb)));

  trialSymbs.addAll(0, production);
  p.production = SUBsymb;

  return parse(trialSymbs, trialToks); }
```

- Assumes `TrialPROG p` has just been removed the symbol list
- Makes copies of symbol & token list, for the trial matching

Some code for top-down parsing

How to do trial productions

PROG $\rightarrow$ `stmt` *// ... a stub for something more complex*

Some code for top-down parsing

How to do trial productions

PROG $\rightarrow$ **`stmt`** *// ... a stub for something more complex*

- First, define a class for the **`stmt`** productions:

```
class PROGproductionStmt extends PROG {
  PROG compact() { return this; }
  PROGproductionStmt () { /* nothing to do here */ } }
```

Some code for top-down parsing

How to do trial productions

PROG → **`stmt`** *// ... a stub for something more complex*

- First, define a class for the **`stmt`** productions:

```
class PROGproductionStmt extends PROG {
  PROG compact() { return this; }
  PROGproductionStmt () { /* nothing to do here */ } }
```

- Then, define a function to try parsing the symbol list using this production:

```
bool tryPROGproductionStmt(TrialPROG p, List<Symb> symbs,
                                        List<Token> toks) {
  List<Symb> trialSymbs = new LinkedList<>(symbs);
  List<Symb> trialToks = new LinkedList<>(toks);
  trialSymbs.add(0, new Stmt());
  p.production = new PROGproductionStmt ();

  return parse(trialSymbs, trialToks); }
```

Some code for top-down parsing

How to do trial productions

PROG → **`begin`** SUB
| **`stmt`** *// ... a stub for something more complex*

Some code for top-down parsing

How to do trial productions

PROG → **begin** SUB
| **stmt** *// ... a stub for something more complex*

A procedure to try all possible productions:

```
bool tryPROGproductions (TrialPROG p, List<Symb> symbs,
                                      List<Token> toks) {
  bool succeeded = tryPROGproductionSUB(p, symbs, toks);
  if (! succeeded)
    succeeded = tryPROGproductionStmt(p, symbs, toks);

  if (succeeded)
    p.production = p.production.compact();

  return succeeded; }
```

Some code for top-down parsing

How to do trial productions

PROG → **`begin`** SUB
| **`stmt`** *// ... a stub for something more complex*

A procedure to try all possible productions:

```
bool tryPROGproductions (TrialPROG p, List<Symb> symbs,
                                      List<Token> toks) {
  bool succeeded = tryPROGproductionSUB(p, symbs, toks);
  if (! succeeded)
    succeeded = tryPROGproductionStmt(p, symbs, toks);

  if (succeeded)
    p.production = p.production.compact();

  return succeeded; }
```

— literally just tries all productions, until it either succeeds or runs out of productions. (And: does a bit of cleanup if a production leads to a successful parse.)

Some code for top-down parsing

How to do trial productions

$$\text{SUB} \rightarrow \text{PROG } \texttt{;} \text{ SUB}$$

Some code for top-down parsing

How to do trial productions

$$\text{SUB} \rightarrow \text{PROG ; SUB}$$

Define a class for the **PROG** production:

```
class SUBproductionPROG extends SUB {
  PROG left; SUB right;

  SUBproductionPROG (PROG p, SUB s) {
    left = p; right = s; }

  SUB compact () {
    left = left.compact(); right = right.compact();
    return this; } }
```

Some code for top-down parsing

How to do trial productions

$$\text{SUB} \rightarrow \text{PROG ; SUB}$$

Define a class for the **`PROG`** production:

```
class SUBproductionPROG extends SUB {
  PROG left; SUB right;

  SUBproductionPROG (PROG p, SUB s) {
    left = p; right = s; }

  SUB compact () {
    left = left.compact(); right = right.compact();
    return this; } }
```

. . . the only part of the grammar with much complexity

- left and right children in the parse tree
- the **`compact()`** method also obtains compact forms of both sub-trees.

Some code for top-down parsing

How to do trial productions

$$\text{SUB} \rightarrow \text{PROG } \texttt{;} \text{ SUB}$$

Some code for top-down parsing

How to do trial productions

$$\text{SUB} \rightarrow \text{PROG ; SUB}$$

Define a function for trial substitutions for this production:

```
bool trySUBproductionPROG(TrialSUB s, List<Symb> symbs,
                                      List<Token> toks) {
  List<Symb> trialSymbs = new LinkedList<>(symbs);
  List<Symb> trialToks = new LinkedList<>(toks);
  Symb PROGsymb = new TrialPROG ();
  Token semicTok = new Semic ();
  Symb SUBsymb  = new TrialSUB ();
  List<Symb> production = new LinkedList<>(
    Arrays.asList(PROGsymb, semicTok, SUBsymb)));

  trialSymbs.addAll(0, production);
  s.production = new SUBproductionPROG (PROGsymb, SUBsymb);

  return parse(trialSymbs, trialToks); }
```

— again using a call to a **`parse`** function (still to be defined).

Some code for top-down parsing

How to do trial productions

$$\text{SUB} \rightarrow \texttt{end}$$

A procedure to test this production for a SUB variable at the beginning of a symbol list:

Some code for top-down parsing

How to do trial productions

$$\text{SUB} \rightarrow \texttt{end}$$

A procedure to test this production for a SUB variable at the beginning of a symbol list:

- First, define a class for the **`end`** productions:

```
class SUBproductionEnd extends SUB {
  SUB compact() { return this; }
  SUBproductionEnd () { /* nothing to do here */ } } }
```

Some code for top-down parsing

How to do trial productions

$$\text{SUB} \rightarrow \texttt{end}$$

A procedure to test this production for a SUB variable at the beginning of a symbol list:

- First, define a class for the **end** productions:

```
class SUBproductionEnd extends SUB {
  SUB compact() { return this; }
  SUBproductionEnd () { /* nothing to do here */ } } }
```

- Then, define a function to try parsing the symbol list using this production:

```
bool tryPROGproductionStmt(TrialSUB s, List<Symb> symbs,
                                       List<Token> toks) {
  List<Symb> trialSymbs = new LinkedList<>(symbs);
  List<Symb> trialSymbs = new LinkedList<>(toks);
  trialSymbs.add(0, new End());
  s.production = new SUBproductionEnd ();

  return parse(trialSymbs, trialToks); }
```

Some code for top-down parsing

How to do trial productions

$$\begin{aligned} \text{SUB} &\rightarrow \text{PROG } \texttt{;} \text{ SUB} \\ &\mid \texttt{end} \end{aligned}$$

Some code for top-down parsing

How to do trial productions

$$\begin{aligned} \text{SUB} &\rightarrow \text{PROG } \texttt{;} \text{ SUB} \\ &\mid \texttt{end} \end{aligned}$$

A procedure to try all possible productions:

```
bool trySUBproductions (TrialSUB s, List<Symb> symbs,
                                    List<Token> toks) {
  bool succeeded = trySUBproductionPROG(s, symbs, toks));
  if (! succeeded)
    succeeded = trySUBproductionEnd(s, symbs, toks));

  if (succeeded)
    s.production = s.production.compact();

  return succeeded; }
```

Some code for top-down parsing

How to do trial productions

$$\text{SUB} \rightarrow \text{PROG } \texttt{;} \text{ SUB} \mid \texttt{end}$$

A procedure to try all possible productions:

```
bool trySUBproductions (TrialSUB s, List<Symb> symbs,
                                    List<Token> toks) {
  bool succeeded = trySUBproductionPROG(s, symbs, toks));
  if (! succeeded)
    succeeded = trySUBproductionEnd(s, symbs, toks));

  if (succeeded)
    s.production = s.production.compact();

  return succeeded; }
```

— again, just tries all productions until it either succeeds or runs out of productions. (And again, does a bit of cleanup if successful.)

Some code for top-down parsing

Token matching

We define a simple subroutine to match tokens from the token list, against tokens in the symbol list (by removing them from the fronts of both lists):

Some code for top-down parsing

Token matching

We define a simple subroutine to match tokens from the token list, against tokens in the symbol list (by removing them from the fronts of both lists):

```
bool matchTokens ( List<Symb> symbs, List<Token> toks )
  bool nextSymbolIsToken = true;
  Symb curSymbol = new Symb();
  Token curToken = new Token();

  if (toks.isEmpty() || symbs.isEmpty())
    return (toks.isEmpty() && symbs.isEmpty());
  nextSymbolIsToken = symbs.get(0).isToken();

  while (nextSymbolIsToken) {
    curSymbol = symbs.remove(0);
    Token curToken = toks.remove(0);
    if (!curSymbol.isSameType(curToken)) return false;
    if (toks.isEmpty() || symbs.isEmpty())
      return (toks.isEmpty() && symbs.isEmpty());
    nextSymbolIsToken = symbs.get(0).isToken(); }

    return !tokenMismatch; }
```

Some code for top-down parsing

At last: the `parse` function

We may describe ‘top down parsing’ without referring to the particular productions for each non-terminal.

```
bool parse ( List<Symb> symbs, List<Token> toks ) {
  Symb curSymbol = new Symb();

  if (!matchTokens(symbs, toks)) return false;
  if (symbs.isEmpty() && toks.isEmpty()) return true;

  curSymbol = symbs.remove(0);
  if (curSymbol instanceOf TrialPROG) {
    TrialPROG p = (TrialPROG) curSymbol;
    return tryPROGproductions(p, symbs, toks); }
  else if (curSymbol instanceOf TrialSUB) {
    TrialSUB s = (TrialSUB) curSymbol;
    return trySUBproductions(s, symbs, toks);}
  else
    return false; }
```

Some code for top-down parsing

At last: the `parse` function

We may describe 'top down parsing' without referring to the particular productions for each non-terminal.

```
bool parse ( List<Symb> symbs, List<Token> toks ) {
  Symb curSymbol = new Symb();

  if (!matchTokens(symbs, toks)) return false;
  if (symbs.isEmpty() && toks.isEmpty()) return true;

  curSymbol = symbs.remove(0);
  if (curSymbol instanceOf TrialPROG) {
    TrialPROG p = (TrialPROG) curSymbol;
    return tryPROGproductions(p, symbs, toks); }
  else if (curSymbol instanceOf TrialSUB) {
    TrialSUB s = (TrialSUB) curSymbol;
    return trySUBproductions(s, symbs, toks);}
  else
    return false; }
```

— as we described: match tokens as much as possible, and then try productions for any variable at front of the symbol list.

Parsing PROG & SUB productions

Example input:

```
LinkedList<Token> toks
  = new LinkedList(Arrays.asList(
      new Begin(), new Stmt(), new Semic(),
      new Stmt(), new Semic(), new End() ) );
```

Usage:

```
PROG ast = new TrialPROG();
LinkedList<Symb> symbs
  = new LinkedList(Arrays.asList(ast));

if ( parse(symbs, toks) ) {
  ast = ast.compress();
  System.out.println( "success" ); }
else
  System.out.println( "failure" );
```

— result upon success: **`ast`** contains an abstract syntax tree!

Left-recursion

There is one hitch to watch out for. We illustrate with a *very* simple grammar for expressions:

$$E \rightarrow E \texttt{+} E \mid \texttt{num}$$

Left-recursion

There is one hitch to watch out for. We illustrate with a *very* simple grammar for expressions:

$$E \rightarrow E \texttt{ + } E \mid \texttt{num}$$

Following the approach we used above,
if the first production that the parser tried for E is "E + E",
we would get stuck in an infinite loop of substitutions,
without producing a token to match against the input tokens.

Left-recursion

There is one hitch to watch out for. We illustrate with a *very* simple grammar for expressions:

$$E \rightarrow E \texttt{ + } E \mid \texttt{num}$$

Following the approach we used above,
if the first production that the parser tried for E is "E + E",
we would get stuck in an infinite loop of substitutions,
without producing a token to match against the input tokens.

In general: a grammar is **left-recursive** if we can find a variable X and a string y such that

$$X \Rightarrow \cdots \Rightarrow X\,y.$$

Left-recursion

There is one hitch to watch out for. We illustrate with a *very* simple grammar for expressions:

$$E \rightarrow E \texttt{ + } E \mid \texttt{num}$$

Following the approach we used above,
if the first production that the parser tried for E is "E + E",
we would get stuck in an infinite loop of substitutions,
without producing a token to match against the input tokens.

In general: a grammar is **left-recursive** if we can find a variable X and a string y such that

$$X \Rightarrow \cdots \Rightarrow X\,y.$$

There is a danger of getting caught in infinite loops with left-recursive grammars, in a similar way to what we've seen above (albeit possibly involving more recursive calls).

Left-recursion

and how to deal with it

We can re-express left recursive productions in terms of a (non-left-recursive) prefix followed by a suffix.

Strategy. Take any left-recursive variable with reductions

$$X \to X\,y_1 \mid \cdots \mid X\,y_k \mid z_1 \mid \cdots \mid z_n$$

where the z_i don't start with X.

Left-recursion

and how to deal with it

We can re-express left recursive productions in terms of a (non-left-recursive) prefix followed by a suffix.

Strategy. Take any left-recursive variable with reductions

$$X \;\to\; X\,y_1 \;\mid\; \cdots \;\mid\; X\,y_k \;\mid\; z_1 \;\mid\; \cdots \;\mid\; z_n$$

where the z_i don't start with X. These are the **base cases** in the production of strings matching X, which may be extended by adding strings y_a, y_b, *etc.*:

$$X \;\Rightarrow\; z_1 \;\mid\; \cdots \;\mid\; z_n \;\mid\; z_1 y_1 \;\mid\; \cdots \;\mid\; z_i y_a \;\mid\; \cdots \;\mid\; z_i y_a y_b \;\mid\; \cdots$$

Left-recursion

and how to deal with it

We can re-express left recursive productions in terms of a (non-left-recursive) prefix followed by a suffix.

Strategy. Take any left-recursive variable with reductions

$$X \to X y_1 \mid \cdots \mid X y_k \mid z_1 \mid \cdots \mid z_n$$

where the z_i don't start with X. These are the **base cases** in the production of strings matching X, which may be extended by adding strings y_a, y_b, *etc.*:

$$X \Rightarrow z_1 \mid \cdots \mid z_n \mid z_1 y_1 \mid \cdots \mid z_i y_a \mid \cdots \mid z_i y_a y_b \mid \cdots$$

This suggests that we rewrite the reductions for X as

$$\begin{aligned} X &\to z_1 X' \mid \cdots \mid z_n X' \\ X' &\to \varepsilon \mid y_1 X' \mid \cdots \mid y_k X' \end{aligned}$$

Left-recursion

and how to deal with it

We can re-express left recursive productions in terms of a (non-left-recursive) prefix followed by a suffix.

Strategy. Take any left-recursive variable with reductions

$$X \to X\,y_1 \mid \cdots \mid X\,y_k \mid z_1 \mid \cdots \mid z_n$$

where the z_i don't start with X. These are the **base cases** in the production of strings matching X, which may be extended by adding strings y_a, y_b, *etc.*:

$$X \Rightarrow z_1 \mid \cdots \mid z_n \mid z_1 y_1 \mid \cdots \mid z_i y_a \mid \cdots \mid z_i y_a y_b \mid \cdots$$

This suggests that we rewrite the reductions for X as

$$\begin{aligned} X &\to z_1 X' \mid \cdots \mid z_n X' \\ X' &\to \varepsilon \mid y_1 X' \mid \cdots \mid y_k X' \end{aligned}$$

More generally: for "indirect" left-recursion $X \Rightarrow^* X\,y$ — expand rules for X to obtain direct left-recursion, and apply this construction.

Removing left-recursion

Example. Instead of the grammar

$$E \rightarrow E \texttt{+} E \mid \texttt{num}$$

we may consider the grammar

$$\begin{aligned} E &\rightarrow \texttt{num}\, E' \\ E' &\rightarrow \varepsilon \mid \texttt{+}\, E' \end{aligned}$$

Removing left-recursion

Example. Instead of the grammar

$$E \;\to\; E \texttt{ + } E \;\Big|\; \texttt{num}$$

we may consider the grammar

$$\begin{aligned} E &\to \texttt{num}\; E' \\ E' &\to \varepsilon \;\Big|\; \texttt{+}\; E' \end{aligned}$$

— which we *could* further simplify to $E \;\to\; \texttt{num} \;\Big|\; \texttt{num + } E$.

Removing left-recursion

Example. Instead of the grammar

$$E \;\to\; E \texttt{ + } E \;\Big|\; \texttt{num}$$

we may consider the grammar

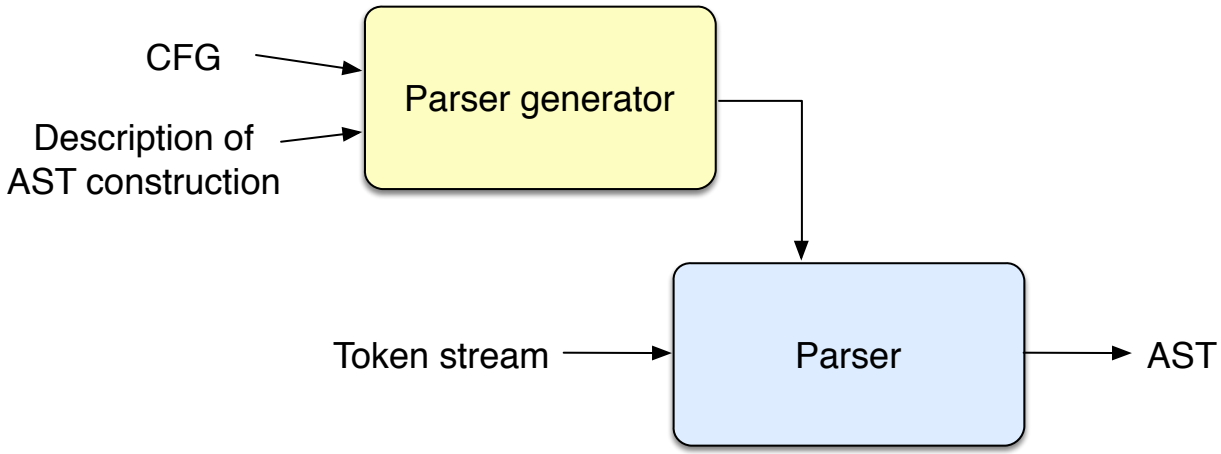
$$\begin{aligned} E &\to \texttt{num}\; E' \\ E' &\to \varepsilon \;\Big|\; \texttt{+}\; E' \end{aligned}$$

— which we *could* further simplify to $E \;\to\; \texttt{num} \;\Big|\; \texttt{num + } E$.

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 in this way.

Parser and lexer generators

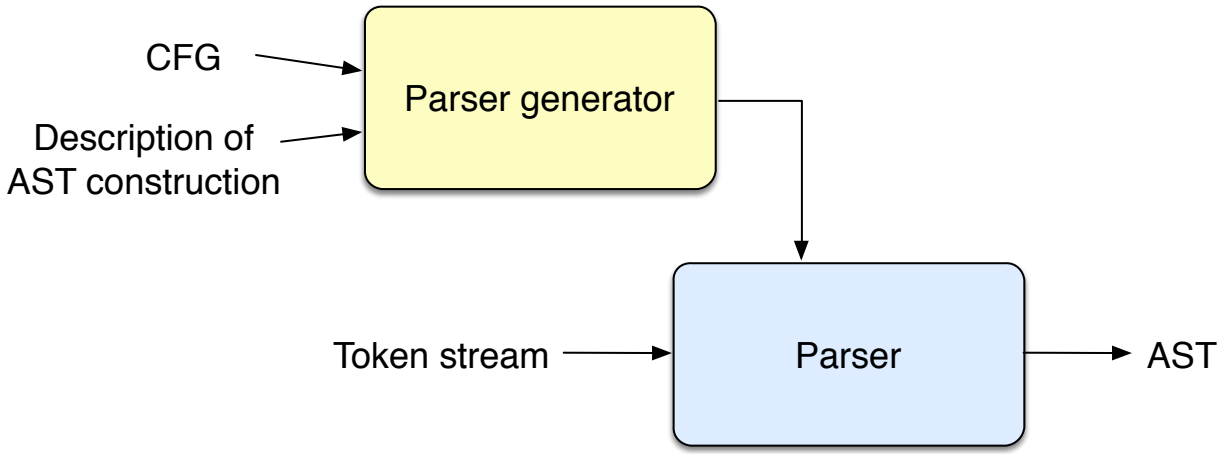
Great news: the construction of a top-down (and also bottom-up) parser is **completely mechanical** given a CFG.



They are available for most programming languages.

Parser and lexer generators

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



They are available for most programming languages.

- Lexers can be generated by **lexer generators** (e.g. JFlex).
- Parsers can be generated by **parser generators** (e.g. JavaCC, CUPS, Yacc, Bison).
- Systems such as BNFC and ANTLR can generate both!

Conclusion

Parsing serves to construct the AST (= abstract syntax tree) of a program, to facilitate code generation:

Conclusion

Parsing serves to construct the AST (= abstract syntax tree) of a program, to facilitate code generation:

- Programming language syntax is given by a CFG (= context-free grammar)

  — together with a specification of the *lexemes* for tokenisation

Conclusion

Parsing serves to construct the AST (= abstract syntax tree) of a program, to facilitate code generation:

- Programming language syntax is given by a CFG (= context-free grammar)

  — together with a specification of the *lexemes* for tokenisation

- CFGs also implicitly provide a specification of how to extract an AST from a source program:
  - for example, written manually for a top-down parser;
  - but also using automatic parser generators.

Conclusion

Parsing serves to construct the AST (= abstract syntax tree) of a program, to facilitate code generation:

- Programming language syntax is given by a CFG (= context-free grammar)

  — together with a specification of the *lexemes* for tokenisation

- CFGs also implicitly provide a specification of how to extract an AST from a source program:
  - for example, written manually for a top-down parser;
  - but also using automatic parser generators.
  - to do so, issues such as ambiguity or left-recursiveness of the CFG might have to be addressed.

Conclusion

Parsing serves to construct the AST (= abstract syntax tree) of a program, to facilitate code generation:

- Programming language syntax is given by a CFG (= context-free grammar)

  — together with a specification of the *lexemes* for tokenisation

- CFGs also implicitly provide a specification of how to extract an AST from a source program:
  - for example, written manually for a top-down parser;
  - but also using automatic parser generators.
  - to do so, issues such as ambiguity or left-recursiveness of the CFG might have to be addressed.
- Parsers are either top-down (easy to write but require absence of left recursion) or bottom-up (more complex).
  - It's usually best to use lexer / parser generators to build lexers / parsers.

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