Lexical Analysis and Parsing

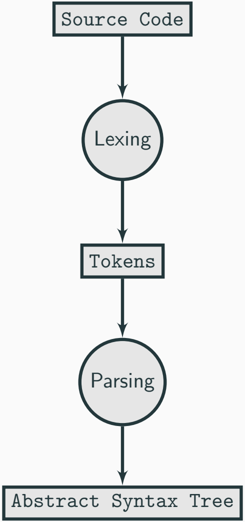
6CCS3COM Computational Models

Josh Murphy

Contents

- **Lexical analysis**
    - Regular expressions
- **Parsing**
    - Top-down parsing (recursive descent parsing)
    - Bottom-up parsing (shift-reduce)

Compilation Process



Lexical Analysis

Tasks for lexical analysis

- An analyser reads a string and coverts it to a **sequence** of **tokens**.
  - Given a sequence of words, which language does each belongs to?
  - Can be done by an abstract machine!

- In practice, an analyser also:
  - remove whitespace (once recognised);
  - remove comments (once recognised);
  - (limited) error detection and correction;
  - marco expansion.

Lexical analysis: examples

- **Arithmetic:**
  - $3 + 4.5 * 67 \Rightarrow$ `NUM(3) PLUSOP NUM(4.5) MULTOP NUM(67)`
- **Human languages:**
  - "Eats shoots and leaves"
  - `VERB(eats) VERB(shoots) AND VERB(leaves)?`
  - `VERB(eats) NOUN(shoots) AND NOUN(leaves)?`
- **Programming languages:**
  - Integer myNum = 4.5;
  - `TYPE(Integer) VAR(myNum) EQUAL FLOAT(4.5) CLOSE`

Tokens of a programming language

- **Keywords:** `if, else, while, do, int`
- **Operators:** `+ - * / !  || ++`
- **Punctuation:** `( ) ; { }`
- **Variables names:** `myNum`
- **Literals:** ‘‘`hello world!`’’
- **Constants:** `42, 3.1415, 1.2e-3, 0x4D1`

Errors in lexical analysis

- **What if a word doesn't belong to any token language?**
    - Can attempt to make modifications until the word fits
    - *e.g.* remove a character, add a character, swap characters...
    - $42. \Rightarrow 42.0, 42, 4.2?$
- **What if a word belongs to more than one token language?**
    - Multiple sequences of tokens are produced.
- **What if the token order doesn't make sense?**
    - A lexical analyser can't recognise this, as it has only a localised view, and does not understand the wider syntactical structure.
    - This is the job of the **parser**.

Defining tokens

- How to define tokens?
    - Explicit sets
    - Abstract machines: *e.g.* finite automata
    - Compact notations: *e.g.* **regular expressions**

Regular expressions

- A regular expression r denotes a language $L(r)$.
  - Recall, a language is a set of acceptable words
- Formation rules for regular expressions over alphabet $\mathcal{X}$:
  - ϵ denotes $\{\epsilon\}$
  - a denotes $\{a\}$, for any symbol $a \in \mathcal{X}$
  - Suppose r and s are regular expressions:
    - $(r)|(S)$ denotes $L(r) \cup L(s)$
    - $(r)?$ denotes $(r)|\epsilon$
    - $(r)(s)$ denotes $L(r)L(s)$
    - $(r)*$ abbreviates $\epsilon|r|r(r)*$
    - $(r)+$ abbreviates $(r)(r)*$
  - $[a - z]$ abbreviates $a|b|c|...|z$

Regular expressions: simple examples

Regular Expression r	Language L(r)
aba	{ aba }
a | (bb) | (aba)	{ aba, bb, aba }
a^*	{ ϵ, a, aa, aaa, ... }
a^+	{ a, aa, aaa, ... }
(ab)+	{ ab, abab, abab, ... }

- **Exercise:** Write a regular expression to represent the variable names of a programming language

Some algebraic rules for regular expressions

$$r|s = s|r$$
$$(r|s)|t = r|s|t$$
$$(rs)t = r(st)$$
$$r(s|t) = rs|rt$$
$$\epsilon r = r$$
$$r\epsilon = r$$
$$r^* = (r|\epsilon)^*$$
$$r^{**} = r^*$$

Regular definitions

- A **regular definition**, over an alphabet $\mathcal{X}$, is a sequence of definitions:

$$\begin{gathered}\langle d_0 \rangle \to r_0 \\ \langle d_1 \rangle \to r_1 \\ \dots \\ \langle d_n \rangle \to r_n\end{gathered}$$

  where each d_i is a distinct name, and each r_i is a regular expression over $\mathcal{X} \cup \{d_0, ..., d_{i-1}\}$

- Regular expressions over $\mathcal{X}$ are called **terminals**.
- $\{d_0, ..., d_n\}$, and regular expression over them, are called **non-terminals**.

Regular definition: example

$$
\begin{aligned}
\langle \texttt{digit} \rangle &\rightarrow 0|1|...|9 \\
\langle \texttt{digits} \rangle &\rightarrow (\langle \texttt{digit} \rangle)\ \langle \texttt{digit} \rangle^* \\
\langle \texttt{fraction} \rangle &\rightarrow \texttt{.}\langle \texttt{digits} \rangle \mid \epsilon \\
\langle \texttt{exponent} \rangle &\rightarrow (\mathrm{e}(+|\text{-})?\langle \texttt{digits} \rangle)? \\
\langle \texttt{number} \rangle &\rightarrow \langle \texttt{digits} \rangle\ \langle \texttt{fraction} \rangle\ \langle \texttt{exponent} \rangle
\end{aligned}
$$

- **Exercises:**
  - What are some valid `number` tokens?
  - Rewrite the rules above so that they are clearer and more concise (apply some algebraic rules and abbreviations).
  - Can all languages be represented by regular expressions?

Regular expressions vs. finite automata

- Regular expressions and finite automata are equivalent forms for representing languages.
  - They can both represent **regular languages** (level-3 of the Chomsky hierarchy).
  - See Kleene's theorem for a proof.

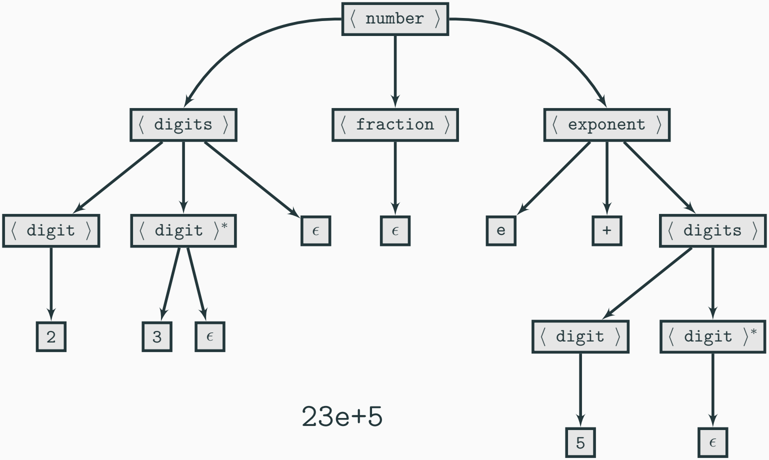
- **Exercise:** recall your regular expression for recognising variables in a programming language; draw its equivalent automaton.

Parsing

Tasks for parsing

- An **parser** reads a sequence of tokens and converts it to an **abstract syntax tree**.
  - Given a sequence of tokens, does their order make sense?
- In practice, a parser also:
  - broader error detection and correction.
- Two types of parsers:
  - Top-down: **Recursive descent parsing (RDP)**
  - Bottom-up: **Shift-reduce**

Abstract syntax trees



Top-down parsing

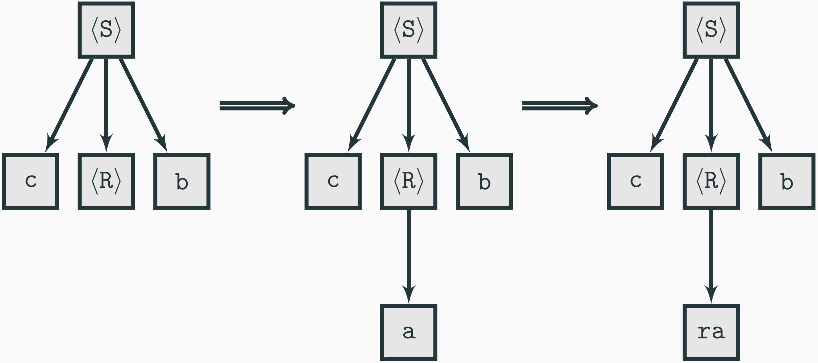
- Starting from the entire string, search for a rule which rewrites the non-terminals to yield terminals consistent with the input.
- May require **back-tracking** if the wrong rewriting rule is selected.

Top-down parsing: example

- Given the following regular definition:

$$\langle \mathrm{S} \rangle \rightarrow cAb$$
$$\langle \mathrm{R} \rangle \rightarrow a|ra$$

Use top-down parsing to determine if `crab` is valid.



Top-down parsing

- How do we decide which rewrite rule to apply when multiple fit?
  - **Recursive Descent Parsing (RDP)** applies the rules from left-to-right.
- Two problems with RDP:
  - **Left recursion**
  - **Repeated terminal checking**

RDP: left recursion example

- **Exercise:** Given the following regular definition:

$$\begin{aligned} \langle \mathtt{T} \rangle &\rightarrow \langle \mathrm{T} \rangle \times \langle \mathrm{D} \rangle \mid \langle \mathrm{D} \rangle \\ \langle \mathtt{D} \rangle &\rightarrow [0-9] \end{aligned}$$

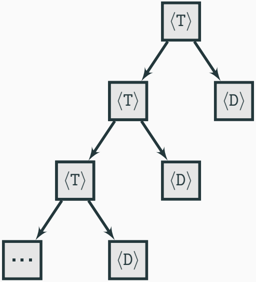
  Use RDP to determine if `9 × 4 × 2` is valid by drawing the abstract syntax tree.

RDP: left recursion example

- **Exercise:** Given the following regular definition:

$$\begin{aligned} \langle \mathtt{T} \rangle &\to \langle \mathrm{T} \rangle \times \langle \mathrm{D} \rangle \mid \langle \mathrm{D} \rangle \\ \langle \mathtt{D} \rangle &\to [0 - 9] \end{aligned}$$

  Use RDP to determine if $9 \times 4 \times 2$ is valid by drawing the abstract syntax tree.



RDP: left recursion

- Left recursion can cause RDP to get stuck in an **infinite loop**.
- A grammar is **left-recursive** if it has a non-terminal A such that there is a derivation $A \to A\alpha$, for some string α.
- Before applying RDP to a grammar we need to **eliminate** left recursion.

RDP: eliminating left recursion

- Can we just flip the order of the terms so the grammar is right recursive?

$$\begin{aligned} \langle \mathtt{T} \rangle \quad &\rightarrow \langle \mathsf{T} \rangle \times \langle \mathsf{D} \rangle \mid \langle \mathsf{D} \rangle \\ \langle \mathtt{D} \rangle \quad &\rightarrow [0-9] \end{aligned}$$

$$\begin{aligned} \langle \mathtt{T} \rangle \quad &\rightarrow \langle \mathsf{D} \rangle \times \langle \mathsf{T} \rangle \mid \langle \mathsf{D} \rangle \\ \langle \mathtt{D} \rangle \quad &\rightarrow [0-9] \end{aligned}$$

- Do these two grammars recognise the same language?
    - Yes...

RDP: eliminating left recursion

- Can we just flip the order of the terms so the grammar is right recursive?

$$\langle \mathtt{S} \rangle \quad \rightarrow \langle \mathtt{S} \rangle \mathsf{a} \mid \mathsf{ba}$$

$$\langle \mathtt{S} \rangle \quad \rightarrow \mathsf{a} \langle \mathtt{S} \rangle \mid \mathsf{ba}$$

- Do these two grammars recognise the same language?
  - No! (*e.g.* `aba`)
- **Some rules don't allow us to just flip to right recursion!**

RDP: eliminating left recursion

- To eliminate left recursion from **any rule**, we can apply the following substitution.

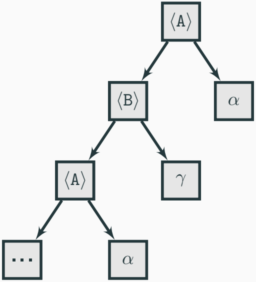
$$\langle \texttt{A} \rangle \quad \rightarrow \langle \texttt{A} \rangle \alpha \mid \beta$$

$$\begin{aligned} \langle \texttt{A} \rangle \quad & \rightarrow \beta \langle \texttt{A'} \rangle \\ \langle \texttt{A'} \rangle \quad & \rightarrow \epsilon \mid \alpha \langle \texttt{A'} \rangle \end{aligned}$$

RDP: indirect left recursion

- We can also run into infinite loops if the grammar is **indirectly left-recursive**.

$$\begin{aligned} \langle \texttt{A} \rangle \quad &\rightarrow \langle \texttt{B} \rangle \alpha \mid \beta \\ \langle \texttt{B} \rangle \quad &\rightarrow \langle \texttt{A} \rangle \gamma \mid \delta \end{aligned}$$



RDP: eliminating indirect left-recursion

- We can **rewrite** indirect left-recursion as an equivalent direct left-recursion by using the substitution rule below. We can then eliminate the left-recursion as before.

$$\frac{\begin{aligned} \langle \mathtt{A} \rangle \quad &\to \langle \mathtt{B} \rangle \alpha \mid \beta \\ \langle \mathtt{B} \rangle \quad &\to \langle \mathtt{A} \rangle \gamma \mid \delta \end{aligned}}{\begin{aligned} \langle \mathtt{A} \rangle \quad &\to (\langle \mathtt{A} \rangle \gamma \mid \delta)\ \alpha \mid \beta \\ \langle \mathtt{A} \rangle \quad &\to \langle \mathtt{A} \rangle (\gamma \alpha) \mid (\delta \alpha \mid \beta) \end{aligned}}$$

RDP: Repeated terminal checking

- Backtracking is always a risk with RDP.
- However, we can make backtracking less expensive by applying **left factoring** to the grammar.
    - This reduces the number of times a token needs to be checked.

RDP: Repeated terminal checking example

- **Exercise:** Given the following regular definition:

$$\begin{aligned} \langle \texttt{R} \rangle &\rightarrow \langle \texttt{S} \rangle\ \langle \texttt{S} \rangle \\ \langle \texttt{S} \rangle &\rightarrow \mathsf{ab} \mid \mathsf{ac} \end{aligned}$$

  Use RDP to determine if `acac` is valid by drawing the abstract syntax tree.

- **Exercise:** Given the following regular definition:

$$\begin{aligned} \langle \texttt{R} \rangle &\rightarrow \langle \texttt{S} \rangle\ \langle \texttt{S} \rangle \\ \langle \texttt{S} \rangle &\rightarrow \mathsf{a}\ \langle \texttt{T} \rangle \\ \langle \texttt{T} \rangle &\rightarrow \mathsf{b} \mid \mathsf{c} \end{aligned}$$

  Use RDP to determine if `acac` is valid by drawing the abstract syntax tree.

RDP: Left factoring

- If the grammar is not left factored we may have to **check terminals more times than necessary**.
- To left factor a grammar we can use the following substitution.
  - Expand rules to expose the common part, and factor it out.

$$\frac{\langle \texttt{A} \rangle \quad \to \alpha\beta \mid \alpha\gamma}{\begin{aligned} \langle \texttt{A} \rangle \quad &\to \alpha\langle \texttt{A'} \rangle \\ \langle \texttt{A'} \rangle \quad &\to \beta \mid \gamma \end{aligned}}$$

RDP: summary

- Easy to construct by hand (if you have a heuristic to guide you to correct terminals)
- However, in order for RDP to be efficient you may have to modify the grammar to remove left recursion, and to left factor.
    - Can be an expensive process.

- So, what about **bottom-up parsing**?

Bottom-up parsing

- In bottom-up parsing, the input is compared against the right-hand sides of the rules, to find where a string can be replaced by a non-terminal.
- Parsing succeeds when the whole input has been replaced.
- We will use the **shift-reduce** parsing techniques.

Shift-reduce example

$$\begin{array}{lll} \text{Rule A:} & \langle \texttt{stat} \rangle & \rightarrow \texttt{begin} \ \langle \texttt{statlist} \rangle \mid \texttt{S} \\ \text{Rule B:} & \langle \texttt{statlist} \rangle & \rightarrow \texttt{end} \mid \langle \texttt{stat} \rangle \ ; \ \langle \texttt{statlist} \rangle \end{array}$$

Is `begin S; S; end` valid?

Shift-reduce example

Rule A: $\langle \texttt{stat} \rangle \rightarrow \texttt{begin} \ \langle \texttt{statlist} \rangle \mid \texttt{S}$

Rule B: $\langle \texttt{statlist} \rangle \rightarrow \texttt{end} \mid \langle \texttt{stat} \rangle \ ; \ \langle \texttt{statlist} \rangle$

Stack	Current Symbol	Remaining Input	Action
	begin	S; S; end	shift

Shift-reduce example

$$\begin{array}{lll} \text{Rule A:} & \langle \texttt{stat} \rangle & \rightarrow \texttt{begin}\ \langle \texttt{statlist} \rangle \mid \texttt{S} \\ \text{Rule B:} & \langle \texttt{statlist} \rangle & \rightarrow \texttt{end} \mid \langle \texttt{stat} \rangle\ ;\ \langle \texttt{statlist} \rangle \end{array}$$

Stack	Current Symbol	Remaining Input	Action
	begin	S; S; end	shift
begin	<u>S</u>	; S; end	reduce (A)

Shift-reduce example

$$\begin{array}{lll} \text{Rule A:} & \langle \texttt{stat} \rangle & \rightarrow \texttt{begin} \ \langle \texttt{statlist} \rangle \mid \texttt{S} \\ \text{Rule B:} & \langle \texttt{statlist} \rangle & \rightarrow \texttt{end} \mid \langle \texttt{stat} \rangle \ ; \ \langle \texttt{statlist} \rangle \end{array}$$

Stack	Current Symbol	Remaining Input	Action
	begin	S; S; end	shift
begin	$\underline{\text{S}}$	; S; end	reduce (A)
begin	stat	; S; end	shift

Shift-reduce example

Rule A: $\langle \texttt{stat} \rangle \rightarrow \texttt{begin}\ \langle \texttt{statlist} \rangle \mid \texttt{S}$

Rule B: $\langle \texttt{statlist} \rangle \rightarrow \texttt{end} \mid \langle \texttt{stat} \rangle\ ;\ \langle \texttt{statlist} \rangle$

Stack	Current Symbol	Remaining Input	Action
	begin	S; S; end	shift
begin	$\underline{\text{S}}$	; S; end	reduce (A)
begin	stat	; S; end	shift
begin stat	;	S; end	shift

Shift-reduce example

$$\begin{array}{lll} \text{Rule A:} & \langle \texttt{stat} \rangle & \rightarrow \texttt{begin} \ \langle \texttt{statlist} \rangle \mid \texttt{S} \\ \text{Rule B:} & \langle \texttt{statlist} \rangle & \rightarrow \texttt{end} \mid \langle \texttt{stat} \rangle \ ; \langle \texttt{statlist} \rangle \end{array}$$

Stack	Current Symbol	Remaining Input	Action
	begin	S; S; end	shift
begin	$\underline{\text{S}}$	; S; end	reduce (A)
begin	stat	; S; end	shift
begin stat	;	S; end	shift
begin stat;	$\underline{\text{S}}$	; end	reduce (A)

Shift-reduce example

$$\begin{array}{lll} \text{Rule A:} & \langle \texttt{stat} \rangle & \rightarrow \texttt{begin}\ \langle \texttt{statlist} \rangle \mid \texttt{S} \\ \text{Rule B:} & \langle \texttt{statlist} \rangle & \rightarrow \texttt{end} \mid \langle \texttt{stat} \rangle\ ;\ \langle \texttt{statlist} \rangle \end{array}$$

Stack	Current Symbol	Remaining Input	Action
	begin	S; S; end	shift
begin	$\underline{\text{S}}$	; S; end	reduce (A)
begin	stat	; S; end	shift
begin stat	;	S; end	shift
begin stat;	$\underline{\text{S}}$	; end	reduce (A)
begin stat;	stat	; end	shift

Shift-reduce example

Rule A: $\langle \texttt{stat} \rangle \rightarrow \texttt{begin}\ \langle \texttt{statlist} \rangle \mid \texttt{S}$

Rule B: $\langle \texttt{statlist} \rangle \rightarrow \texttt{end} \mid \langle \texttt{stat} \rangle\ ;\ \langle \texttt{statlist} \rangle$

Stack	Current Symbol	Remaining Input	Action
	begin	S; S; end	shift
begin	<u>S</u>	; S; end	reduce (A)
begin	stat	; S; end	shift
begin stat	;	S; end	shift
begin stat;	<u>S</u>	; end	reduce (A)
begin stat;	stat	; end	shift
begin stat; stat	;	end	shift

Shift-reduce example

$$\begin{array}{lll} \text{Rule A:} & \langle \texttt{stat} \rangle & \rightarrow \texttt{begin}\ \langle \texttt{statlist} \rangle \mid \texttt{S} \\ \text{Rule B:} & \langle \texttt{statlist} \rangle & \rightarrow \texttt{end} \mid \langle \texttt{stat} \rangle\ ;\ \langle \texttt{statlist} \rangle \end{array}$$

Stack	Current Symbol	Remaining Input	Action
	begin	S; S; end	shift
begin	<u>S</u>	; S; end	reduce (A)
begin	stat	; S; end	shift
begin stat	;	S; end	shift
begin stat;	<u>S</u>	; end	reduce (A)
begin stat;	stat	; end	shift
begin stat; stat	;	end	shift
begin stat; stat;	<u>end</u>		reduce (B)

Shift-reduce example

Rule A: $\langle \texttt{stat} \rangle \rightarrow \texttt{begin}\ \langle \texttt{statlist} \rangle \mid \texttt{S}$

Rule B: $\langle \texttt{statlist} \rangle \rightarrow \texttt{end} \mid \langle \texttt{stat} \rangle\ ;\ \langle \texttt{statlist} \rangle$

Stack	Current Symbol	Remaining Input	Action
	begin	S; S; end	shift
begin	$\underline{\text{S}}$	; S; end	reduce (A)
begin	stat	; S; end	shift
begin stat	;	S; end	shift
begin stat;	$\underline{\text{S}}$	; end	reduce (A)
begin stat;	stat	; end	shift
begin stat; stat	;	end	shift
begin stat; stat;	$\underline{\text{end}}$		reduce (B)
begin stat; $\underline{\text{stat;}}$	$\underline{\text{statlist}}$		reduce (B)

Shift-reduce example

Rule A: $\langle \texttt{stat} \rangle \rightarrow \texttt{begin}\ \langle \texttt{statlist} \rangle \mid \texttt{S}$

Rule B: $\langle \texttt{statlist} \rangle \rightarrow \texttt{end} \mid \langle \texttt{stat} \rangle\ ;\ \langle \texttt{statlist} \rangle$

Stack	Current Symbol	Remaining Input	Action
	begin	S; S; end	shift
begin	S	; S; end	reduce (A)
begin	stat	; S; end	shift
begin stat	;	S; end	shift
begin stat;	S	; end	reduce (A)
begin stat;	stat	; end	shift
begin stat; stat	;	end	shift
begin stat; stat;	end		reduce (B)
begin stat; stat;	statlist		reduce (B)
begin stat;	statlist		reduce (B)

Shift-reduce example

Rule A: $\langle\texttt{stat}\rangle \rightarrow \texttt{begin}\ \langle\texttt{statlist}\rangle \mid \texttt{S}$

Rule B: $\langle\texttt{statlist}\rangle \rightarrow \texttt{end} \mid \langle\texttt{stat}\rangle\ ;\ \langle\texttt{statlist}\rangle$

Stack	Current Symbol	Remaining Input	Action
	begin	S; S; end	shift
begin	S	; S; end	reduce (A)
begin	stat	; S; end	shift
begin stat	;	S; end	shift
begin stat;	S	; end	reduce (A)
begin stat;	stat	; end	shift
begin stat; stat	;	end	shift
begin stat; stat;	end		reduce (B)
begin stat; stat;	statlist		reduce (B)
begin stat;	statlist		reduce (B)
begin	statlist		reduce (A)

Shift-reduce example

Rule A: $\langle\texttt{stat}\rangle \rightarrow \texttt{begin}\ \langle\texttt{statlist}\rangle \mid \texttt{S}$

Rule B: $\langle\texttt{statlist}\rangle \rightarrow \texttt{end} \mid \langle\texttt{stat}\rangle\ ;\ \langle\texttt{statlist}\rangle$

Stack	Current Symbol	Remaining Input	Action
	begin	S; S; end	shift
begin	<u>S</u>	; S; end	reduce (A)
begin	stat	; S; end	shift
begin stat	;	S; end	shift
begin stat;	<u>S</u>	; end	reduce (A)
begin stat;	stat	; end	shift
begin stat; stat	;	end	shift
begin stat; stat;	<u>end</u>		reduce (B)
begin stat; <u>stat;</u>	<u>statlist</u>		reduce (B)
begin <u>stat;</u>	<u>statlist</u>		reduce (B)
<u>begin</u>	<u>statlist</u>		reduce (A)
	stat		accept

Shift-reduce summary

- Shift-reduce is hard to do by hand.
- But efficient (since no need to rewrite the grammar).
- Commonly used in practice.