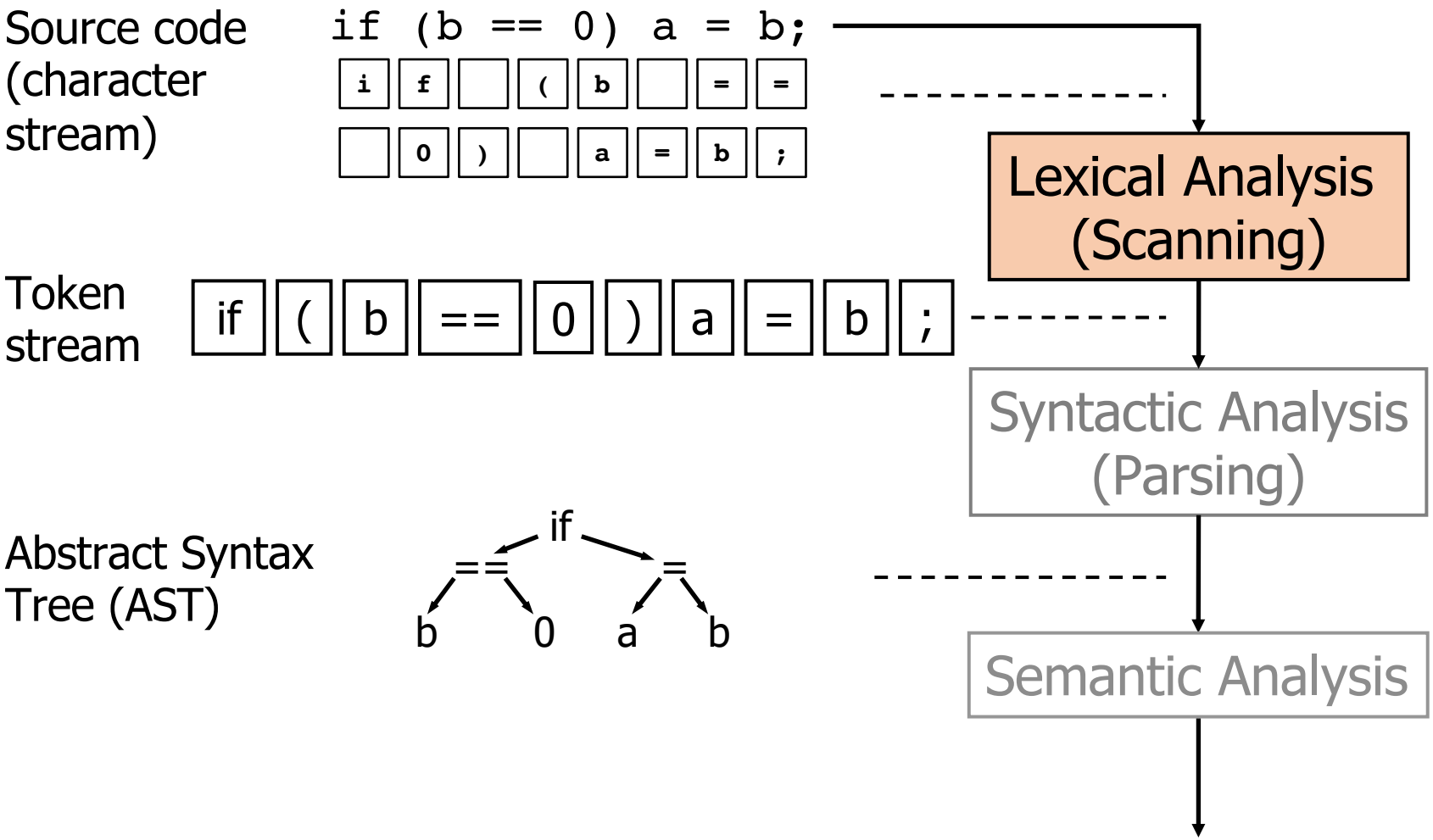
Where We Are


Source code (character stream)
if (b == 0) a = b;
i f (b = =
0) a = b ;
Lexical Analysis (Scanning)
Token stream
if (b == 0) a = b ;
Syntactic Analysis (Parsing)
Abstract Syntax Tree (AST)
if
==
=
b
0
a
b
Semantic Analysis

What The Lexer Does

- Scan the input character stream representing the source program and convert it into a stream of tokens for use by the parser.

- Handle comments and whitespace.
- Track source coordinates to correlate error messages with source text.
- Implement some preprocessing functions.

Tokens, Patterns, Lexemes

- Token
  - The output of the lexer, and a terminal symbol in the parser's grammar.
  - E.g., **ID** for identifiers, **KWD** for keywords.
- Pattern
  - A rule describing a set of input strings that map to the same token.
  - E.g., "an identifier is a letter followed by any number of letters, digits, or underscores".
- Lexeme
  - A (maximal) sequence of source characters that matches the pattern for a token.
  - E.g., `pi`, `D2`, `this_token`.

Tokens, Patterns, Lexemes

- Token
  - The output of the lexer, and a terminal symbol in the parser's grammar.
  - E.g., **ID** for identifiers, **KWD** for keywords.
- Pattern
  - A rule describing a set of input strings that map to the same token.
  - E.g., "an identifier is a letter followed by any number of letters, digits, or underscores".
- Lexeme
  - A (maximal) sequence of source characters that matches the pattern for a token.
  - E.g., `pi`, `D2`, `this_token`.

```
1:if a=0 then a:=b; (* entry := default *)
```

Regular Languages

- A *regular language* is a particularly simple class of patterns describing the structure of tokens in programming languages.
  - Also known as type-3 languages in the Chomsky hierarchy.
  - Typically described compactly using *regular expression* notation.

Regular Languages

- A *regular language* is a particularly simple class of patterns describing the structure of tokens in programming languages.
  - Also known as type-3 languages in the Chomsky hierarchy.
  - Typically described compactly using *regular expression* notation.

- Generated by *regular grammars*, which are a proper subclass of context-free grammars.

Regular Languages

- A *regular language* is a particularly simple class of patterns describing the structure of tokens in programming languages.
  - Also known as type-3 languages in the Chomsky hierarchy.
  - Typically described compactly using *regular expression* notation.

- Generated by *regular grammars*, which are a proper subclass of context-free grammars.

- Recognized by *finite automata*, which are a very restricted form of Turing Machines.

Example: Floating-Point Literals in Java

- Java® Language Specification Java SE 16 Edition, 2021-02-12, § 3.10.2.

A *floating-point literal* has the following parts: a whole-number part, a decimal or hexadecimal point (represented by an ASCII period character), a fraction part, an exponent, and a type suffix.

A floating-point literal may be expressed in decimal (base 10) or hexadecimal (base 16).

For decimal floating-point literals, at least one digit (in either the whole number or the fraction part) and either a decimal point, an exponent, or a float type suffix are required. All other parts are optional. The exponent, if present, is indicated by the ASCII letter `e` or `E` followed by an optionally signed integer.

For hexadecimal floating-point literals, at least one digit is required (in either the whole number or the fraction part), and the exponent is mandatory, and the float type suffix is optional. The exponent is indicated by the ASCII letter `p` or `P` followed by an optionally signed integer.

Underscores are allowed as separators between digits that denote the whole-number part, and between digits that denote the fraction part, and between digits that denote the exponent.

A floating-point literal is of type `float` if it is suffixed with an ASCII letter `F` or `f`; otherwise its type is `double` and it can optionally be suffixed with an ASCII letter `D` or `d`.

Floating-Point Literals in Java, Simplified

- Java® Language Specification Java SE 16 Edition, 2021-02-12, § 3.10.2.

A *floating-point literal* has the following parts: a whole-number part, a decimal ~~or hexadecimal~~ point (represented by an ASCII period character), a fraction part, an exponent, and a type suffix.

~~A floating-point literal may be expressed in decimal (base 10) or hexadecimal (base 16).~~

For ~~decimal~~ floating-point literals, at least one digit (in either the whole number or the fraction part) and either a decimal point, an exponent, or a float type suffix are required. All other parts are optional. The exponent, if present, is indicated by the ASCII letter `e` or `E` followed by an optionally signed integer.

~~For hexadecimal floating-point literals, at least one digit is required (in either the whole number or the fraction part), and the exponent is mandatory, and the float type suffix is optional. The exponent is indicated by the ASCII letter `p` or `P` followed by an optionally signed integer.~~

~~Underscores are allowed as separators between digits that denote the whole-number part, and between digits that denote the fraction part, and between digits that denote the exponent.~~

A floating-point literal is of type `float` if it is suffixed with an ASCII letter `F` or `f`; otherwise its type is `double` and it can optionally be suffixed with an ASCII letter `D` or `d`.

A Grammar For This Language

DecimalFloatingPointLiteral →

Digits `.` Digits? ExponentPart? FloatTypeSuffix? |

`.` Digits ExponentPart? FloatTypeSuffix? |

Digits ExponentPart FloatTypeSuffix? |

Digits ExponentPart? FloatTypeSuffix

ExponentPart → ExponentIndicator SignedInteger

ExponentIndicator → `e` | `E`

SignedInteger → Sign? Digits

Sign: → `+` | `-`

FloatTypeSuffix → `f` | `F` | `d` | `D`

Digits → Digit | Digit Digits

Digit → `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

A Regular Expression For This Language

ExponentPart → `[eE]([+-]?)([0-9]+)`

DecimalFloatingPointLiteral →

`[0-9]+ . (([0-9]+)?)` ExponentPart`?` `[fFdD]?` |

`.` `[0-9]+` ExponentPart`?` `[fFdD]?` |

`[0-9]+` ExponentPart `[fFdD]?` |

`[0-9]+` ExponentPart`?` `[fFdD]`

- Standard `grep` syntax
  - *Shortcuts*: `[...]` for enumeration; `[a-b]` for contiguous range;
  - *Set operations*: | for union; juxtaposition for concatenation;
  - *Closure operations*: `?` for zero or one instance; + for one or more instances; `*` for zero or more instances;
  - *Grouping*: `( )` , for clarity only.

Recognizers For Regular Languages

- Recognition: A decision problem.
  - Given a regular expression r over an alphabet Σ and a string $s \in \Sigma^*$, is $s \in L(r)$?

- Such a recognizer is known as a finite automaton.
  - Examines each symbol of s exactly once, in sequence order.
  - After examining each character, updates a summary of the string prefix it has encountered so far in some *fixed finite amount of state*.
  - After completing the scan of the string, either ends up in an *accepting state* (in which case $s \in L(r)$) or in a *non-accepting state* (in which case $s \notin L(r)$).