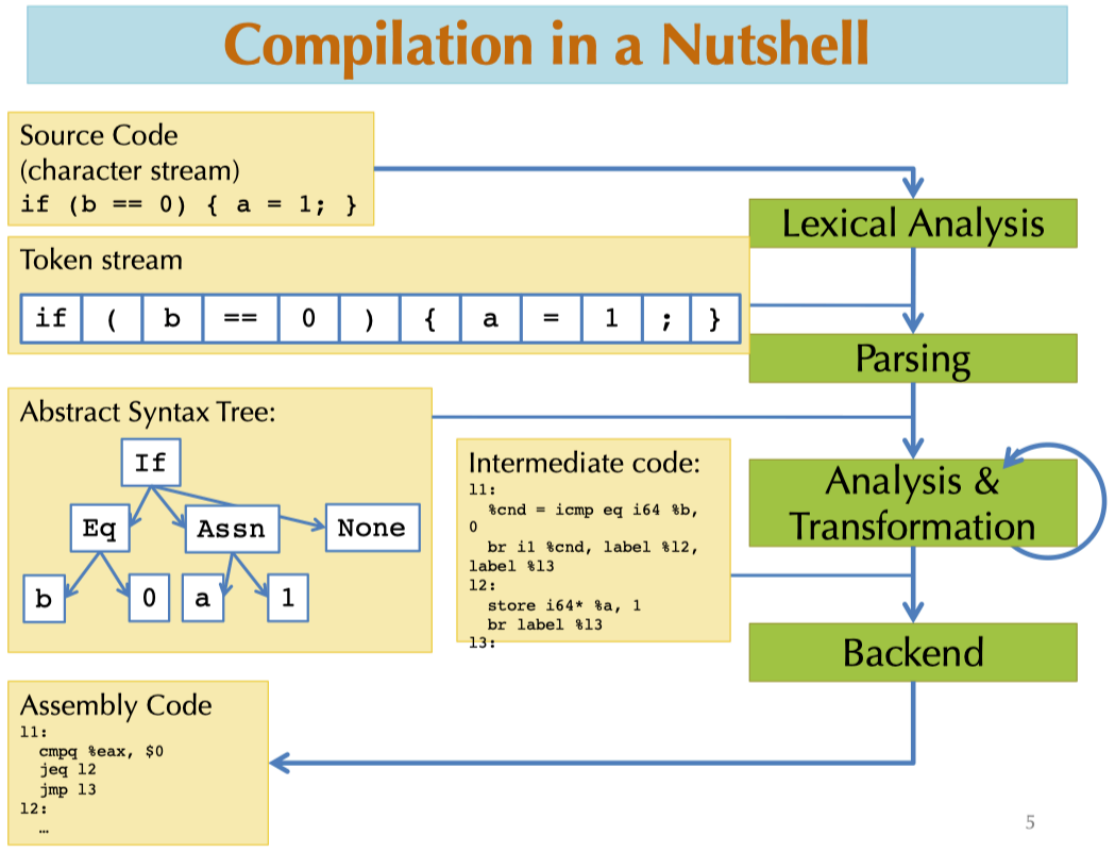
Compiler Design — Lecture notes week 5

- Author: Ruben Schenk
- Date: 24.10.2021
- Contact: ruben.schenk@inf.ethz.ch

6. Lexing

6.1 Compilation in a Nutshell



6.2 Lexical Analysis

The first step of lexing is the **lexical analysis**. Its goal is to change the *character stream*, e.g. "`if (b == 0) a = 1;`" into *tokens:*

```
if (b == 0) a = 1;
=>
IF; LPAREN; Ident("b"); EQEQ; Int(0); RPAREN; LBRACE; Ident("a"); Eq;
Int(1); Semi; RBRACE;
```

A **token** is a data type that represents indivisible chunks of text;

- Identifiers; `a y11 elsex _100`
- Keywords: `if else while`
- Integers: `2 200 -500 5L`
- Floating point: `2.0 .02 1e5`
- Symbols: `+ * { } ( ) ++ << >> >>>`

- Strings: `"x" "He said, \"Are you?\""`
- Comments: `(* Compiler Design: Project 1 ...*)`

Often delimited by *whitespace* (`' ', \t` etc.). In some languages (e.g. Python or Haskell) whitespace is significant!

6.3 Principled Solution to Lexing

6.3.1 Regular Expressions

Regular expressions precisely describe sets of strings. A regular expression `R` has one of the following forms:

- ϵ : stands for the empty string
- `'a'` : an ordinary character stands for itself
- `R_1 | R_2` : alternatives, stands for a choice of `R_1` or `R_2`
- `R_1R_2` : concatenation, stands for `R_1` or `R_2`
- `R*` : Kleene start, stands for zero or more repetitions of `R`

There are some useful extensions to the above-mentioned forms:

- `"foo"` : strings, equivalent to `'f''o''o'`
- `R+` : one or more repetitions of `R`, equivalent to `RR*`
- `R?` : zero or one occurrence of `R`, equivalent to (ϵ | R)
- `['a'-'z']` : one of `a`, `b`, ..., or `z`, equivalent to `(a|b|...|z)`
- `[^'0'-'9']` : any character except `0` through `9`
- `R as x` : name the string matched by `R` as `x`

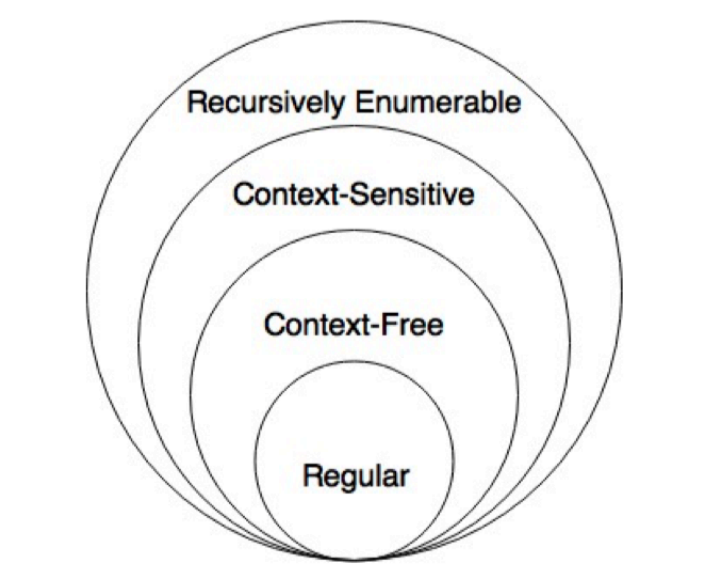
Examples:

- Recognize the keyword "if" : `"if"`
- Recognize a digit: `['0'-'9']`
- Recognize an integer literal: `'-'?['0'-'9']+`

In practice, it is useful to *name* regular expressions:

```
let lowercase = ['a'-'z']
let uppercase = ['A'-'Z']
let character = uppercase | lowercase
```

6.3.2 Chomsky Hierarchy



6.3.3 How to Match?

Consider the input string "`ifx = 0`". We could lex this input as either `'if''x''=''0'` or as `'ifx''=''0'`. Thus, regular expressions alone can be ambiguous. We need a rule for choosing between the options above:

- Most languages choose the **longest match**
- In that case, the second option would be chosen above

We get **conflicts** with tokens whose regular expressions have a shared prefix. How do we resolve this?

- Ties are broken by giving some matches higher priority (example, give keywords priority over identifiers)
- Usually specified by the order the rules appear in the lex input file

6.4 Lexer Generator

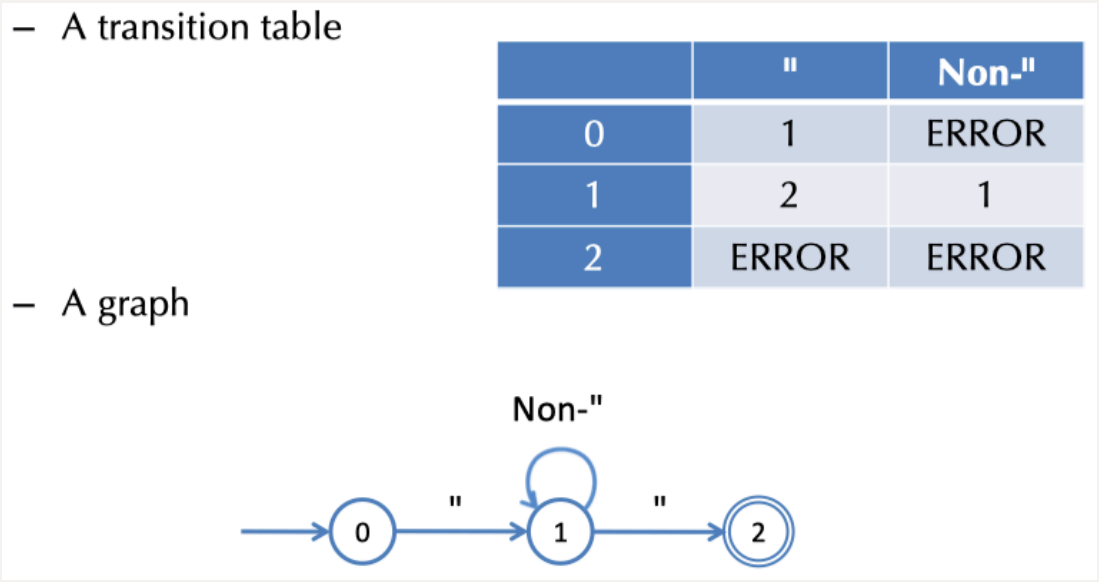
The **lexer generator** reads a list of regular expressions: `R_1, ..., R_n`, one per token.
Each token has an attached *action* `A_i`, which is simply a piece of code to run when the regular expression is matched:

```
1  rule token = parse
2      | '-'?digit+                          { Int (Int32.of_string(lexeme
   lexbuf)) }
3      | '+'                                 { PLUS }
4      | 'if'                                { IF }
5      | character (digit|character|'_')*    { Ident(lexeme lexbuf) }
6      | whitespace+                         { token lexbuf }
```

6.5 Finite Automata

6.5.1 Deterministic Finite Automaton

A **deterministic finite automaton** (DFA) can be represented as:

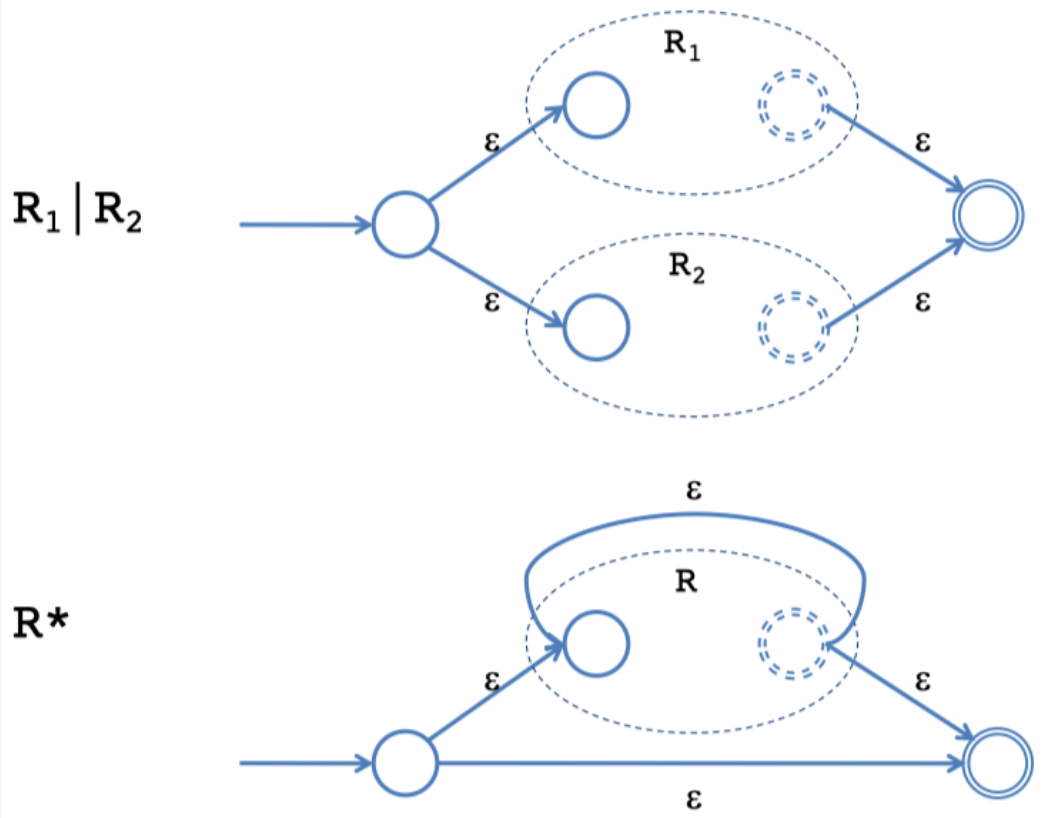


> *We can build a finite automaton for every regular expression!*

6.5.2 Nondeterministic Finite Automaton

A **nondeterministic finite automaton** (NFA) is built the same way as a DFA (i.e. finite set of states, start state, accepting states, transition arrows connecting states, etc.). However, different to DFA's, NFA's can have two arrows leaving the same state with the same label!

Sums and Kleene stars can easily be represented with NFA's:



6.5.3 DFA vs. NFA

- *DFA*
    - action of the automaton for each input is fully determined
    - accepts if the input is consumed upon reaching an accepting state
    - obvious table-based implementation
- *NFA*
    - automaton potentially has a choice at every step
    - accepts an input if there exists a way to reach an accepting state
    - less obvious how to implement efficiently

6.5.4 NFA to DFA conversion

The idea to convert a NFA to a DFA is to run all possible executions of the NFA in "parallel" and meanwhile keep track of a set of possible states (so-called "finite fingers").

Example: Consider `-?[0-9]+`:

6.6 Summary of Lexer Generator Behavior

1. Take each regular expression `R_i` and its action `A_i`
2. Compute the NFA formed by `(R_1 | R_2 | ... | R_n)`
3. Compute the DFA for the big NFA computed in the previous step
4. Compute the minimal equivalent DFA
5. Produce the transition table
6. Implement the longest match
    a. Start from initial state
    b. Follow transitions, remember the last accepted state entered
    c. Accept the input until no transition is possible
    d. Perform the highest-priority action associated with the last accepted state

7. Parsing

Parsing describes the process of finding a syntactic structure, such as an **abstract syntax tree** (AST).

7.1 Overview

Parsing, i.e. syntactic analysis, works as follows:

- Input: stream of tokens
- Output: abstract syntax tree

The strategy is to parse the token stream to traverse the "concrete" syntax. During the traversal, we build a tree representing the "abstract" syntax.

However, for this to work, we need to know how our concrete syntax looks like. In other words, we need to describe our language syntax precisely and conveniently.

7.2 Context Free Grammars

7.2.1 Overview

The idea of **context free grammars** (CFG) is to derive a string in the language starting from some element and rewrite it according to the given rules.

Example: Consider the specification of the language of balances `parens`:

$$\mathrm{S} \to (\mathrm{S})\mathrm{S}$$
$$\mathrm{S} \to \epsilon$$

A derivation could look like:

$$\mathrm{S} \to (\mathrm{S})\mathrm{S} \to ((\mathrm{S})\mathrm{S})\mathrm{S} \to ((\epsilon)\mathrm{S})\mathrm{S} \to ((\epsilon)\mathrm{S})\epsilon \to ((\epsilon)\epsilon)\epsilon = (())$$

7.2.2 CFG's Mathematically

A **context free grammar** (CFG) consists of:

- a set of *terminals*
- a set of *nonterminals*
- a designated nonterminal called the *start symbol*

- a set of *productions* `LHS -> RHS`:
    - LHS is a nonterminal
    - RHS is a string of terminals and nonterminals

Example: A grammar that accepts parenthesized sums of numbers:

$$\mathrm{S} \to \mathrm{E} + \mathrm{S} \,|\, \mathrm{E}$$
$$\mathrm{E} \to \mathrm{number} \,|\, (\mathrm{S})$$

7.2.3 Derivations

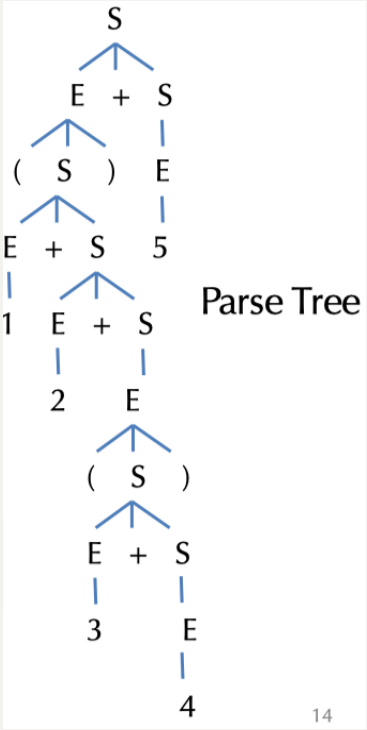
For arbitrary strings α, β, γ and production rule $A \to \beta$, a **single step of derivation** is:

$$\alpha A \gamma \to \alpha \beta \gamma$$

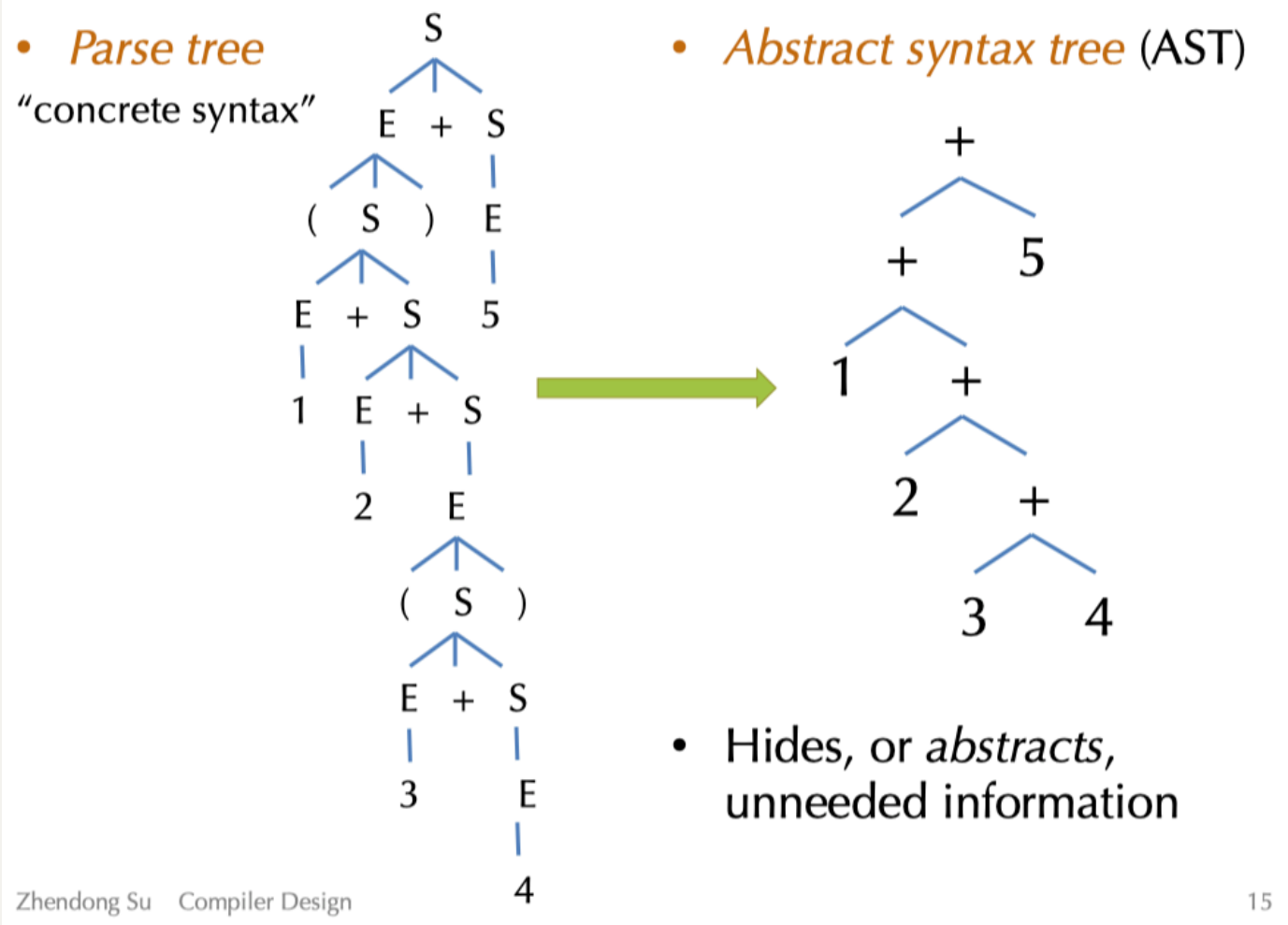
We might represent a derivation as a tree where:

- Leaves: terminals
- Internal nodes: nonterminals

Example: Derivation tree of `(1 + 2 + (3 + 4)) + 5`:



From Parse Trees to Abstract Syntax Trees



Derivation Orders

Production of a grammar can be applied in any order, however, there are two standard orders:

- *Leftmost derivation*: Find the left-most nonterminal and apply a production to it
- *Rightmost derivation*: Find the right-most nonterminal and apply a production there

Remark: Both strategies and any other order yield the same parse tree!

7.2.4 Loops and Termination

Some care is needed when defining CFG's:

$$\mathrm{S} \to \mathrm{E}$$
$$\mathrm{E} \to \mathrm{S}$$

- This grammas has nonterminal definitions that are *non-productive*, i.e. they don't mention any terminal symbols

- There is no finite derivation starting from S, so the language is empty

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

- This grammar is productive, but again there is no finite derivation string from S, so the language is *empty*

7.3 Grammars for Programming Languages

7.3.1 Associativity

Consider the following grammar:

$$\begin{aligned} \mathrm{S} &\to \mathrm{E} + \mathrm{S} \mid \mathrm{E} \\ \mathrm{E} &\to \mathrm{number} \mid (\mathrm{S}) \end{aligned}$$

This grammar makes the `+` *right associative*, i.e. the AST is the same for both `1 + 2 + 3` and `1 + (2 + 4)`. Note also that the grammar is *right recursive* due to the production $\mathrm{S} \to \mathrm{E} + \mathrm{S}$.

How would we make the `+` *left associative*? Simple:

$$\begin{aligned} \mathrm{S} &\to \mathrm{S} + \mathrm{E} \mid \mathrm{E} \\ \mathrm{E} &\to \mathrm{number} \mid (\mathrm{S}) \end{aligned}$$

7.3.2 Ambiguity

Consider the following grammar:

$$\mathrm{S} \to \mathrm{S} + \mathrm{S} \mid (\mathrm{S}) \mid \mathrm{number}$$

This accepts the same set of strings as the previously mentioned grammar. We can get both right and left associativity for the `+` operator.

> ***Non-ambiguous*** *means that for every input string, there is only one way to parse it!*

However, not all operations are associative. Moreover, if there are multiple operations, amiguity in the grammar leads to ambiguity in their *precedence*.

Consider the grammar:

$$\mathrm{S} \to \mathrm{S} + \mathrm{S} \mid \mathrm{S} * \mathrm{S} \mid (\mathrm{S}) \mid \mathrm{number}$$

The input `1 + 2 * 3` might be parsed either:

- `(1 + 2) * 3 = 9`
- `1 + (2 * 3) = 7`

Eliminating Ambiguity

We can often eliminate ambiguity by adding nonterminals and allowing recursion only on the left (or right).

To disambiguate the previously introduced grammar:

- Decide to make `*` higher precedence than `+`
- Make `+` left associative
- Make `*` right associative

Note: `s_2` corresponds to "atomic" expressions:

$$
\begin{aligned}
S_0 &\to S_0 + S_1 \mid S_1 \\
S_1 &\to S_2 * S_1 \mid S_2 \\
S_2 &\to \text{number} \mid (S_0)
\end{aligned}
$$