Lexical analysis

Token

- More abstract than the notion of lexeme
- A token corresponds to some kind of lexemes
- Example: identifiers, numbers, the assignment operator, . . .
- In some cases it can carry semantic information: numbers $\rightsquigarrow$ their values

Tokenizer

A program which performs lexical analysis, recognizes lexemes and generates tokens

Example in C/Java/C++/C#

The string `"x2=042;"` is decomposed in the following tokens:

- `IDENTIFIER` with name `"x2"`
- `ASSIGN_OP`
- `INT_NUMBER` with value thirty-four
- `STATEMENT_TERMINATOR`

Regular languages

Definition

A regular language is a language definable with a regular expression.

Remark: regular languages can be defined in other equivalent ways:

- with *right* or *left regular* (also called *linear*) grammars;
- with non-deterministic or deterministic finite automata (NFA or DFA).

Limitations

Regular languages are quite simple. Examples:

- the language of identifiers
- the language of numbers

Regular expressions can define the syntax of the units that constitute a programming language, but cannot define the syntax of the full language.

Regular languages

Examples of non-regular languages

- The language of expressions with natural numbers, binary addition and multiplication and parentheses cannot be defined by a regular expression.

  Remark: the actual problem are the parentheses; if one removes them, then the language becomes regular!
- A very simple example of non-regular language:
  $\{\texttt{"a"}^n\texttt{"b"}^n \mid n \text{ natural number}\} = \{\texttt{""}, \texttt{"ab"}, \texttt{"aabb"}, \texttt{"aaabbb"}, \ldots\}$.

Syntax analysis

Defined on top of lexical analysis.

Definition

The problem of

- recognizing a sequence of lexemes/tokens as valid according to some syntactic rules
- building, in case of success, an abstract representation of the recognized sequence for performing some computation on it

Parser

A program which performs syntax analysis

Syntax analysis for programming languages (PL)

Parsers for PL

- they recognize sequences of tokens generated by a *tokenizer*
- the syntax rules are formally defined by a *grammar*
- they generate
  - a *parse/derivation tree*: a less abstract representation of the analyzed sequence
  - an *abstract syntax tree (AST)*: a more abstract representation
- they can be
  - either hand-written
  - or automatically generated from a grammar with specific software tools as ANTLR or Bison

A parser at work

Example 1 with C/Java/C++/C# syntax

Analyzed tokens:

- `IDENTIFIER` with name `"x2"`
- `INT_NUMBER` with value thirty-four
- `ASSIGN_OP`
- `STATEMENT_TERMINATOR`

Result of the parser: failure, the sequence is not recognized and one or more error messages are issued.

A parser at work

Example 2 with C/Java/C++/C# syntax

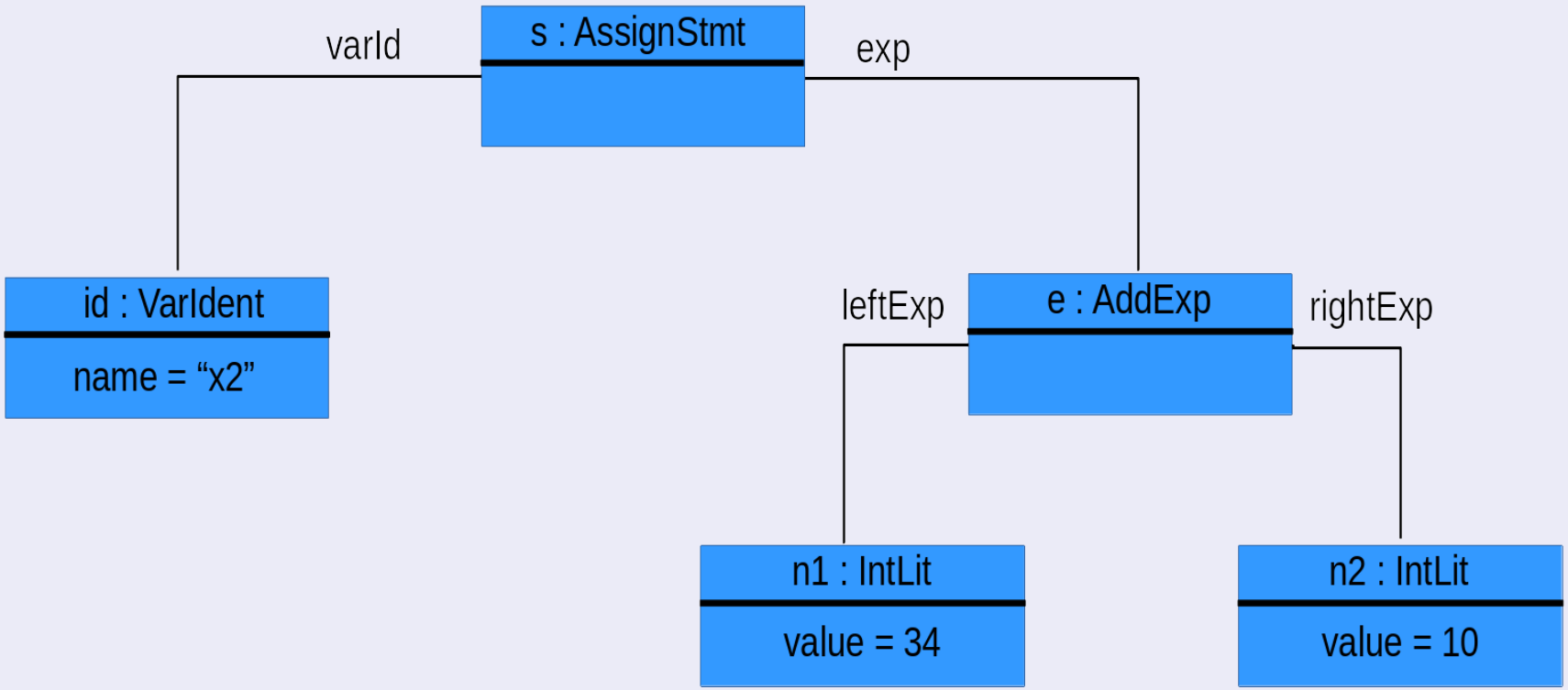
Analyzed tokens:

- `IDENTIFIER` with name `"x2"`
- `ASSIGN_OP`
- `INT_NUMBER` with value thirty-four
- `ADD_OP`
- `INT_NUMBER` with value ten
- `STATEMENT_TERMINATOR`

A parser at work

Example 2 with C/Java/C++/C# syntax

Result of the parser: success, the sequence is recognized and the following AST is generated:



Context free (CF) grammars

In a nutshell

- the most widespread formalism for defining the syntax rules of a PL
- more expressive than regular expressions
- based on concatenation and union plus names and recursive (=inductive) definitions

A first example

A CF grammar for simple expressions:

```
Exp ::= Num | Exp '+' Exp | Exp '*' Exp | '(' Exp ')'
Num ::= '0' | '1'
```