Programming Language Design

Syntactic / Syntax Analysis

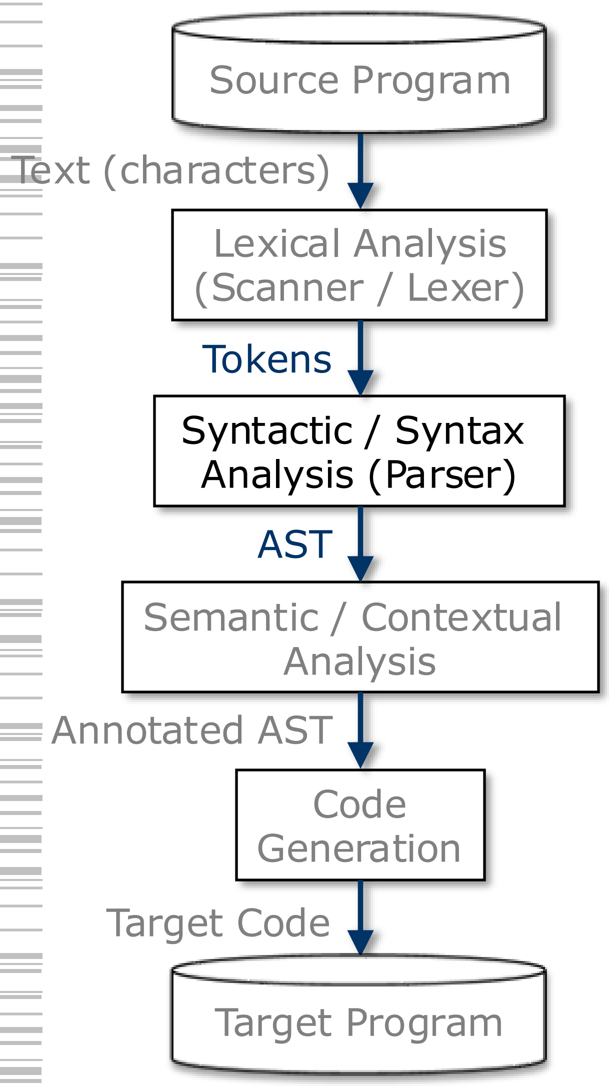
Francisco Ortin

Department of Computer Science

Contents

- **Objectives of the Syntax Analyzer**
- Derivations and Parse Trees
- Ambiguous Grammars
- Parsing Strategies
- The ANTLR Parser Generator
- Abstract Syntax Trees

Objectives



- The **syntactic analyzer** (parser) has two objectives:
  1. Checking whether tokens represent a valid sequence (i.e., the program is syntactically correct), A.K.A. *parsing*
  2. Building an AST representing the structure of the source program
- If the tokens do not conform to the syntax rules of the language, a syntax error is produced
- Abstract Syntax Trees (**AST**s) are simplified parse trees

Example

Source:

```
int a,b;
read b;
a = b+3;
```

Lexical analysis

Tokens:

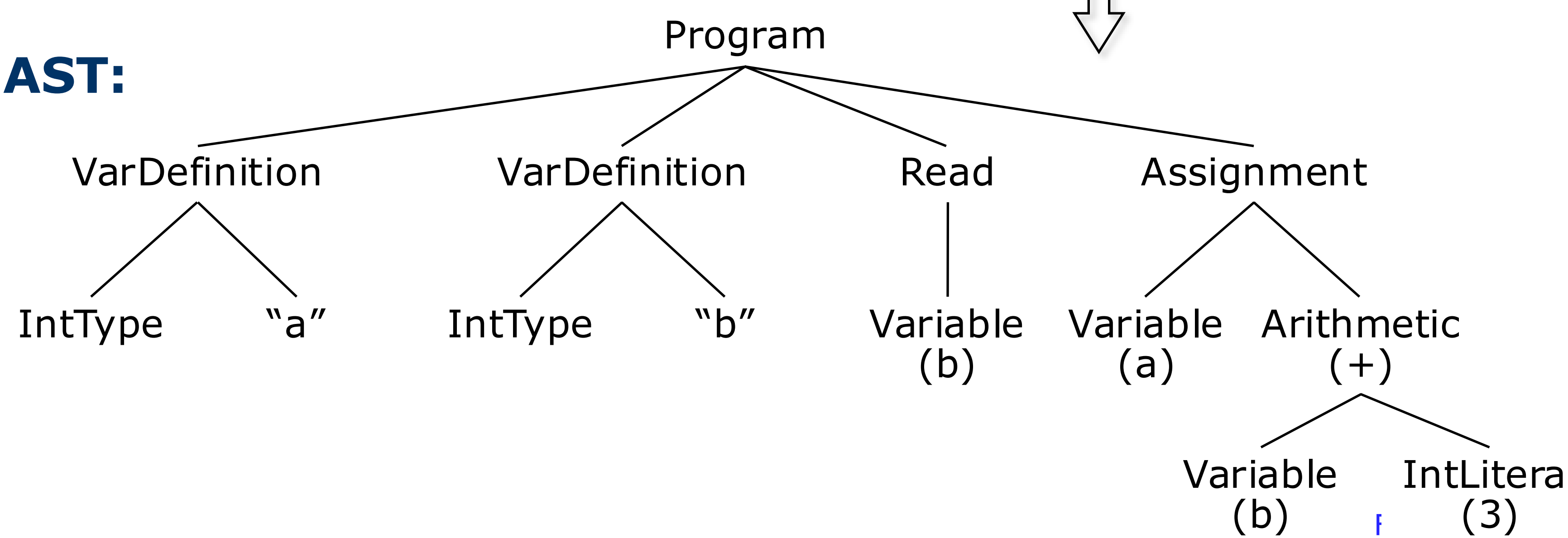
```
INT ID ',' ID ';' READ ID ';' ID '=' ID '+' INT_CONSTANT ';'
```

Syntax analysis

1st The program is syntactically correct

2nd The AST is generated:

AST:



Contents

- Objectives of the Syntax Analyzer
- **Derivations and Parse Trees**
- Ambiguous Grammars
- Parsing Strategies
- The ANTLR Parser Generator
- Abstract Syntax Trees

Grammars

- We will first see how to **parse** programs: checking whether the program is syntactically correct
- The parsing process analyzes whether the source program conforms to the rules of a formal **grammar**

	Language	Grammar	Automaton
Type 0	Recursively enumerable	Unrestricted	Turing machine
Type 1	Context-sensitive	Context-sensitive	Linear bounded automaton
Type 2	**Context-free**	**Context-free**	**Pushdown automaton** ← *Parsers*
Type 3	Regular	Regular	Finite state machine

One-Step Derivations

- Recall (from lexical analysis):
  - A **string** is a sequence of symbols (terminal and non-terminals), i.e., $\alpha \mid \alpha \in (V_N \cup V_T)^*$
    - A **one-step derivation** of a string is the application of one grammar production that transforms the string into another one
    - One-step derivations are denoted by $\Rightarrow$
- A **parser** recognizes a valid sequence of tokens by performing derivations following the grammar productions
- Example:

$$
\begin{aligned}
e &\rightarrow A\ e\ B \\
&\mid\ \varepsilon
\end{aligned}
$$

Derivations			Production applied
e	$\Rightarrow$	A e B	$e \rightarrow A\ e\ B$
	$\Rightarrow$	A A e B B	$e \rightarrow A\ e\ B$
	$\Rightarrow$	A A A e B B B	$e \rightarrow A\ e\ B$
	$\Rightarrow$	A A A B B B	$e \rightarrow \varepsilon$

Derivation

- A **derivation** of a string is a <u>sequence of one-step derivations</u>

$e \rightarrow A\ e\ B$
$\quad |\ \varepsilon$

- A derivation is denoted by $\Rightarrow^*$

  $e \Rightarrow^* AABB \qquad AAeBB \Rightarrow^* AAAAABBBBB$

- Therefore, the language ***L*** defined by a grammar ***G*** can be formalized as

  $L(G) = \{\alpha \in V_T^* : s \Rightarrow^* \alpha \}$

  The α string is commonly called a **sentence** or **program**

- Our <u>example</u> grammar G, $e \rightarrow A\ e\ B \mid \varepsilon$

  defines the language

  $L(G) = \{A^nB^n : n \geq 0 \}$ (comprehension)

  $L(G) = \{\varepsilon, AB, AABB, AAABBB\ldots \}$ (extension)

Mandatory Activity

- Determine (by extension) the language defined by the following grammar:

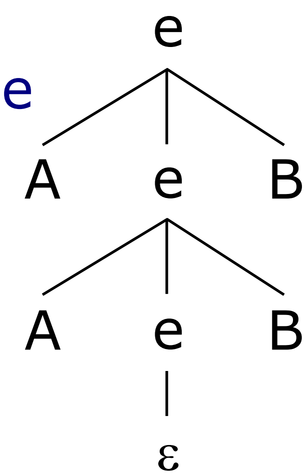
(1)	*stmt*	$\rightarrow$	*if-stmt*
(2)		|	ID = *exp* ;
(3)	*exp*	$\rightarrow$	ID
(4)		|	INT_CONSTANT
(5)	*if-stmt*	$\rightarrow$	IF (*exp*) *stmt*
(6)		|	IF (*exp*) *stmt* ELSE *stmt*

Parse Trees

- **One-step derivations** determine the **process** of program recognition
- However, they do not represent the **structure** of the recognized programs
- For this purpose, derivations can be represented as **tree structures**
- A **parse tree** describes productions applied in each one-step derivation to recognize a program

$$
\boxed{\begin{aligned} e &\rightarrow A\ e\ B \\ &\mid\ \varepsilon \end{aligned}}
$$

Parse Tree



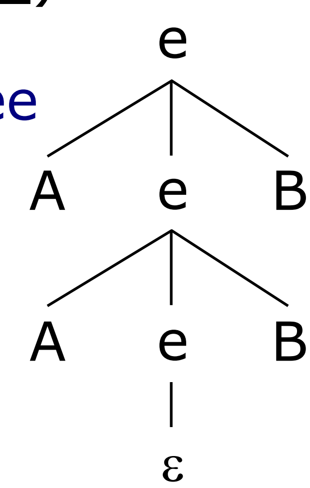
Derivations	Productions applied
$e \Rightarrow A\ e\ B$	$e \rightarrow A\ e\ B$
$\Rightarrow A\ A\ e\ B\ B$	$e \rightarrow A\ e\ B$
$\Rightarrow A\ A\ B\ B$	$e \rightarrow \varepsilon$

Parse Trees

- In parse trees
  - **Parent nodes** represent the (non-terminal) left-hand side symbol of the production used in that one-step derivation
  - **Child nodes** represent the right-hand side symbols of the production used in that one-step derivation
  - The **root node** is the *S* start symbol
  - **Leaf nodes** are terminal symbols (tokens)

$$\boxed{\begin{array}{l} e \rightarrow A\ e\ B \\ \quad\ \mid\ \varepsilon \end{array}}$$

Parse Tree



Derivations	Productions applied
$e \Rightarrow A\ e\ B$	$e \rightarrow A\ e\ B$
$\Rightarrow A\ A\ e\ B\ B$	$e \rightarrow A\ e\ B$
$\Rightarrow A\ A\ B\ B$	$e \rightarrow \varepsilon$

Mandatory Activity

- Given the following CFG grammar

(1) $exp \rightarrow exp$ [exp]

(2) | ID

(3) | INT_CONSTANT

- Represent the parse tree for the 2 following programs

```
v[3]
w[a][5]
```

Contents

- Objectives of the Syntax Analyzer
- Derivations and Parse Trees
- **Ambiguous Grammars**
- Parsing Strategies
- The ANTLR Parser Generator
- Abstract Syntax Trees

Useless Symbols

- ***n*** is a **non-generating** non-terminal symbol if it does not derive a string of terminals

  That is, $n \not\Rightarrow^* \beta$, where $\beta \in V_T{}^*$

  <u>Example</u>:

  $s \rightarrow a \mid n$

  $a \rightarrow \mathrm{A} \mid \varepsilon$

  $\boldsymbol{n} \rightarrow \mathrm{B}\ b\ c$

  $b \rightarrow \mathrm{B} \mid \mathrm{C}$

  $c \rightarrow \mathrm{C}\ n$

Useless Symbols

- ***n*** is a **non-reachable** non-terminal symbol if the start symbol ***s*** does not derive ***n***

  That is, $s \not\Rightarrow^* n$

  Example:

  $s \rightarrow a$ A

  $n \rightarrow$ B a

  $a \rightarrow \varepsilon$ | A B

- If ***n*** is either a **non-generating** or **non-reachable** symbol, it is said to be **useless**
- CFGs must not have useless symbols
  - Otherwise, they are erroneous

Mandatory Activity

- Activity: Given the following grammar

(1)	*stmt*	$\rightarrow$	*if-stmt*
(2)		|	ID = *exp* ;
(3)	*exp*	$\rightarrow$	ID
(4)		|	INT_CONSTANT
(5)	*if-stmt*	$\rightarrow$	IF (*exp*) *stmt*
(6)		|	IF (*exp*) *stmt* ELSE *stmt*

- Is the following program syntactically valid?
- If so, identify the parse tree

```
if (a)
        if (b) c=1;
else   c=2;
```

Ambiguous Grammars

- A grammar that generates two distinct parse trees for the same program is an **ambiguous grammar**
  - Two distinct derivations exist for the same program
- Ambiguous grammars represent a <u>serious problem</u> because the semantics of the distinct trees are (commonly) distinct too
  - Therefore, the generated program may be distinct
- We must not use ambiguous grammars to define a language, because the generated programs may be incorrect
- The decision problem of whether a grammar is ambiguous is **undecidable** (no algorithm exists)

Bibliography

- Alfred V. Aho, Monica S. Lam. Compilers: Principles, Techniques, and Tools. Addison Wesley, 2nd Edition, 2006.
- Terence Parr. The Definitive ANTLR 4 Reference, 2nd edition. Pragmatic Bookshelf, 2013.
- David A. Watt. Programming Language Processors in Java: Compilers and Interpreters. Prentice Hall, 2000.
- Michael L. Scott. Programming Language Pragmatics. Morgan Kaufmann, 4th edition 2015.