Syntax Analysis

Dr. Nguyen Hua Phung
nhphung@hcmut.edu.vn

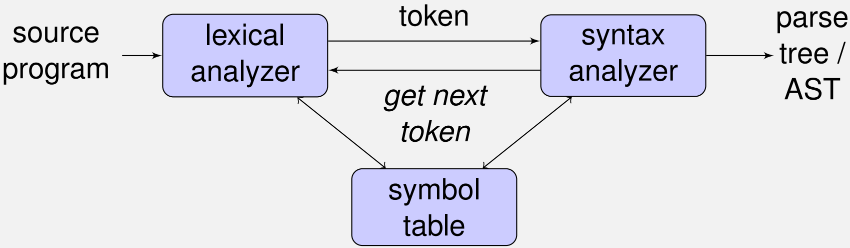
HCMC University of Technology, Viet Nam

07, 2019

Outline

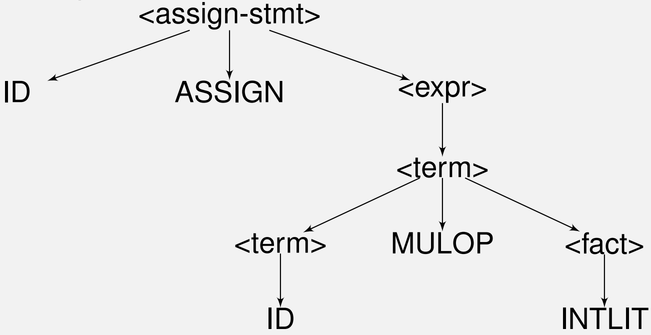
Roles

- read the sequence of tokens
- produce as output a parse tree or abstract syntax tree (AST)
- give error messages when detecting syntax errors



Example

- Source program: a = b * 4
- Lexer output (parser input): ID ASSIGN ID MULOP INTLIT
- Parser output:



Can we use regular expression to express the following language?

$$L = \{a^n b^n \mid n > 0\}$$

Example

In programming languages, there are some symmetric structure

- (((...))) the number of (must be equal to that of)
- **repeat ... repeat ... until ... until**

More generally, programming languages have many recursive structures

Example

<expr> ::= <expr> + <expr>
<stmt> ::= **if** <expr> **then** <stmt> **else** <stmt>

Regular expressions cannot describe this kind of structure

We need

- A mean to describe this kind of language
- A method to detect if a sequence of tokens is valid or invalid regarding to this kind of language

A context-free grammar (CFG) consists of

- A set of terminals T
- A set of non-terminals N
- A start symbol S $\in$ N
- A set of productions P

A production p $\in$ P is in the form: X $\rightarrow \alpha$
where X $\in$ N and α is a sequence of symbols in T and/or N

Example

CFG for simple integer expressions consists of:

- Set of terminals (tokens):{ADDOP, MULOP, INTLIT, LB, RB}
- Set of non-terminals:{<exp>}
- Start symbol: <exp>
- Set of productions:

  <exp> → <exp> ADDOP <exp>
  <exp> → <exp> MULOP <exp>
  <exp> → INTLIT
  <exp> → LB <exp> RB

Example

CFG for simple integer expressions consists of:

- Set of terminals (tokens):{ADDOP, MULOP, INTLIT, LB, RB}
- Set of non-terminals:{<exp>}
- Start symbol: <exp>
- Set of productions:

<exp>	$\rightarrow$	<exp> ADDOP <exp>
	|	<exp> MULOP <exp>
	|	INTLIT
	|	LB <exp> RB

How to know the language which a CFG describe?

Language Generation

1. Start with the string containing only the start symbol
2. Replace any non-terminal symbol X in the string with the right-hand side of some production X $\rightarrow \alpha$
3. Repeat (2) until there are no non-terminals in the string

Grammar

<exp>	$\rightarrow$	<exp> ADDOP <exp>	(1)
	$\vert$	<exp> MULOP <exp>	(2)
	$\vert$	LB <exp> RB	(3)
	$\vert$	INTLIT	(4)

Derivation

<exp>	$\overset{(2)}{\Rightarrow}$	<exp> MULOP <exp>
	$\overset{(4)}{\Rightarrow}$	INTLIT MULOP <exp>
	$\overset{(1)}{\Rightarrow}$	INTLIT MULOP <exp> ADDOP <exp>
	$\overset{(4)}{\Rightarrow}$	INTLIT MULOP INTLIT ADDOP <exp>
	$\overset{(4)}{\Rightarrow}$	INTLIT MULOP INTLIT ADDOP INTLIT

Let G be a context-free grammar with start symbol S. The language L(G) generated from G is:

$$\{ a_1 a_2 ... a_n \mid a_i \in T \text{ and } S \stackrel{+}{\Rightarrow} a_1 a_2 ... a_n \}$$

CFG is a good mean to describe PL, but:

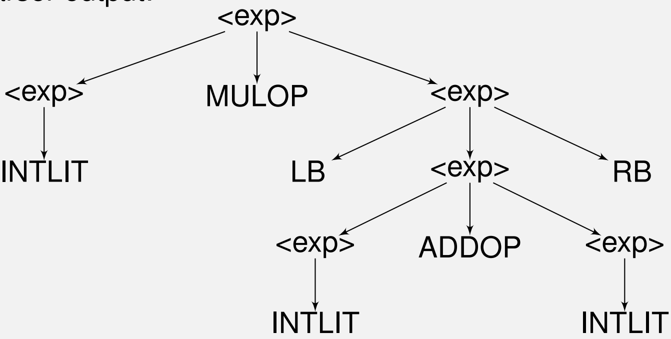
- Not only check if a string is valid but also its parse tree
- how to handle gracefully error

Parse Tree

- Start symbol as the parse tree's root
- For a production X $\rightarrow Y_1 Y_2 ... Y_n$, add children $Y_1 Y_2 ... Y_n$ to node X

Example

- Source: 12 * (4 + 5)
- Parser input: INTLIT MULOP LB INTLIT ADDOP INTLIT RB
- Parser output:

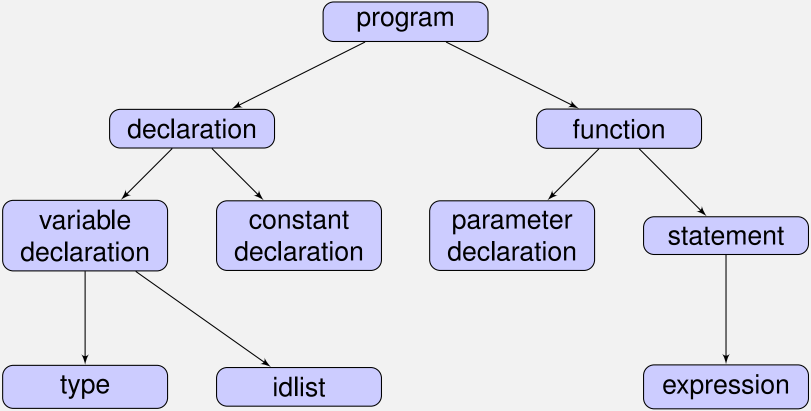


- Based on the language specification
- Try to find out the **hierarchy structure** of a language
- Focus on the **order of syntax units** instead of their meaning or other constraints
- Use **recursion** to describe something occurring **many times**
- Use **recursion** to describe **nested structures**

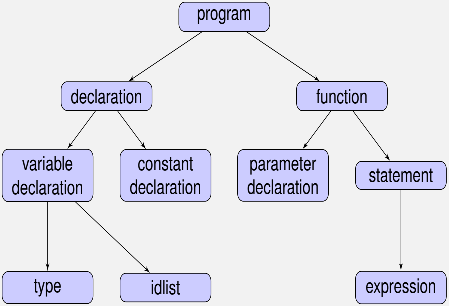
Each language has its own specification which sometimes differs from common sense

- **Smalltalk** has the same precedence for all binary operators
  a + b * c $\equiv$ ((a + b) * c)
- **C** specifies > , < , >= , <= higher priority than ==, != while **Pascal** lets these operators same priority

Hierarchy Structure



Example



$$
\begin{array}{rcl}
\text{program} & \rightarrow & \ldots \\
\text{declaration} & \rightarrow & \ldots \\
\text{variable\_decl} & \rightarrow & \ldots \\
\text{const\_decl} & \rightarrow & \ldots \\
\text{type\_decl} & \rightarrow & \ldots \\
\text{idlist} & \rightarrow & \ldots \\
\text{function} & \rightarrow & \ldots \\
\text{para\_decl} & \rightarrow & \ldots \\
\text{stmt} & \rightarrow & \ldots \\
\text{exp} & \rightarrow & \ldots
\end{array}
$$

Just the order, not other constraints

- CFG just helps to describe the order of tokens
- CFG cannot be used to describe type constraints
- Some other kinds of constraints such as scope, name resolution,... cannot be solved by CFG

2.1 Constant Declaration:

Each constant declaration has the form:
<identifier> = <expression> ;
The *<expression>* will be discussed later in the Expression section. Note that the expression in a constant declaration must be evaluated statically, so it cannot include a variable or a method invocation.
For example:
My1stCons = 1 + 5;
*MyndCons = 2 * My1stCons;*

Some language structures may contain unlimited list of elements:

- A variable declaration may have a list of identifiers
  a, c10, b: integer;
- A function declaration may have a list of parameter
  function foo(**a:integer; b:real**)
- A block may have many statements inside
  begin
  c := 1;
  m := 10;
  end

Write grammar rule for <many>

Use **recursion** to describe <many> based on <one>

<many>	$\rightarrow$	<one> <many'>
	|	<non recursive case>
<many'>	$\rightarrow$	[<separator>] <one> <many'>
	|	ϵ

- <separation>: symbol is used to separate elements in a list
  a, c10, d: integer
  function foo(a:integer;b:real)
- <non recursive case>: the minimum number of elements in the list
  - <one> if there is at least one element
  - ϵ if the list may be empty

For example,

listid	$\rightarrow$	ID tailid	|	ID
tailid	$\rightarrow$	COMMA ID tailid	|	ϵ

In programming languages, there are many nested structures:

- There may be a function declaration in a function declaration
- A statement may appear inside another statement.
- A block may appear inside another block
- An expression may be an operand of another expression
- ...

Use **recursion** to describe a nested structures, for example

<stmt>	$\rightarrow$	IF <exp> THEN <stmt> ELSE <stmt>
	|	WHILE <exp> DO <stmt>
	|	...

Extended Backus-Naur Form(EBNF)

EBNF

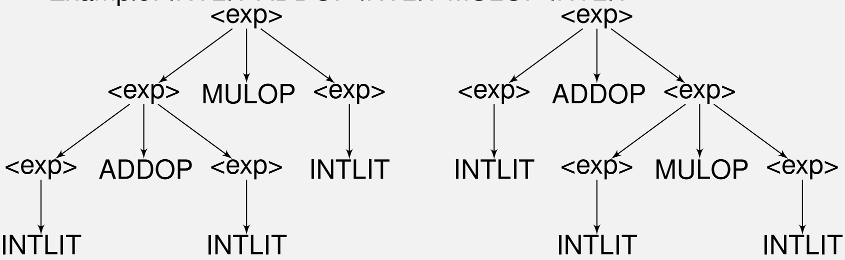
- allow to use operators in regular expression in RHS
- higher expressiveness
- often supported by top-down parsing generators

BNF	EBNF (RHS)	ANTLR
<exp> $\rightarrow$ <exp> '+' <term> $\mid$ <exp> '-' <term> $\mid$ <term>	<term> (('+'|'-') <term>)*	exp: term (('+'|'-') term)*;
<else> $\rightarrow$ ELSE <stmt> $\mid$ ϵ	(ELSE <stmt>)?	else: ("else" stmt)?;
<idlist> $\rightarrow$ ID ',' idlist $\mid$ ID	ID (',' ID)*	idlist: ID ("," ID)* ;

Ambiguous grammar

- When more than one parse tree can be found for a string of tokens, the grammar is *ambiguous*
- Ambiguity makes the meaning of some programs ill-defined

Example: INTLIT ADDOP INTLIT MULOP INTLIT



ADDOP calculated first

MULOP calculated first

- Rewrite the grammar unambiguously
- In some grammar tools, there are disambiguating declarations

<exp> $\rightarrow$ <exp> ADDOP <exp> | <exp> MULOP <exp>
| INTLIT
| LB <exp> RB

- The recursive term <exp> appeared in both sides of ADDOP and MULOP is the source of ambiguity.
- Removing one recursive term in right hand side of productions can disambiguous the grammar.

<exp> $\rightarrow$ <exp> ADDOP <term> | <exp> MULOP <term>
| <term>
<term> $\rightarrow$ INTLIT
| LB <exp> RB

- A binary operator *o* is left-associated (or right-associated) when the left (or right) operator *o* in expression *x o x o x* is calculated first.

For example,
9 - 5 - 2 $\Rightarrow$ 6 $\Rightarrow$ 2

- The operator side where the recursive term appears will determine the association of the operator

For example,

- if ADDOP is left-associated, the grammar should be
  <exp> $\rightarrow$ <exp> ADDOP <term>
- if ADDOP is right-associated, the grammar should be
  <exp> $\rightarrow$ <term> ADDOP <exp>

- To priorize opterators, the left hand sides (lhs) of the rules where these operators appear should be different
- if operator o_1 has higher priority than operator o_2, o_1's lhs is generated by o_2's lhs

For example,

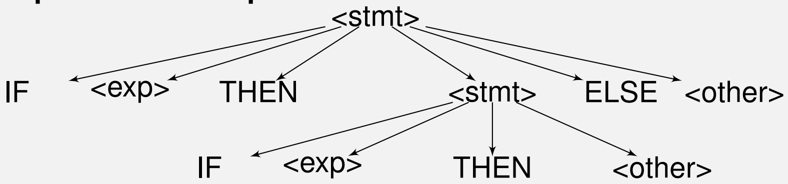
<exp>	$\rightarrow$	<exp> ADDOP <term>
	|	<term>
<term>	$\rightarrow$	<term> MULOP <factor>
	|	<factor>
<factor>	$\rightarrow$	INTLIT
	|	LB <exp> RB

If Statement

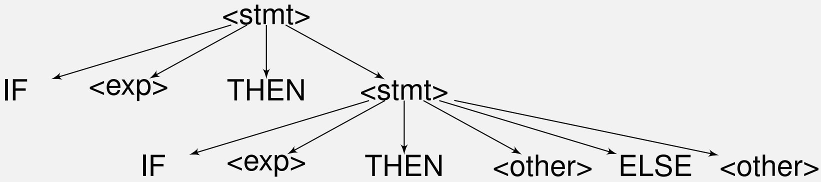
<stmt> → IF <exp> THEN <stmt> ELSE <stmt>
| IF <exp> THEN <stmt>
| <other>

Draw parse tree for input

IF <exp> THEN IF <exp> THEN <other> ELSE <other>



or



Disambiguous grammar of if statement

Natural if-statement grammar

<stmt> $\rightarrow$ IF <exp> THEN <stmt> ELSE <stmt>
| IF <exp> THEN <stmt>
| <other>

Unambiguous if-statement grammar

<stmt> $\rightarrow$ <matchStmt>
| <unmatchStmt>
<matchStmt> $\rightarrow$ IF <exp> THEN <matchStmt> ELSE <matchStmt>
| <other>
<unmatchStmt> $\rightarrow$ IF <exp> THEN <stmt>
| IF <exp> THEN <matchStmt> ELSE <unmatchStmt>

Declarations in grammar tools

- Some grammar tools has declarations to disambiguous grammar
- Write grammar in natural format and use declarations to disambiguous it

For example, ANTLR has

- use option <assoc=left>, <assoc=right> for left- and right-associativity, respectively.
- the order of these declarations makes the order of operator precedence

Example in ANTLR

```
main    :    expr EOL
        ;
expr    :
             LB expr RB
        |    ADDOP expr                        // highest
        |    expr MULOP expr                   //default: left-assoc
        |    expr ADDOP expr
        |    <assoc=right>  expr '=' expr      // lowest, right-assoc
        |    INT
        |    ID
        ;
```

Summary

- Context-free grammar: only one non-terminal symbol on the LHS
- Some issues when writing a CFG
  - ambiguous - dis-ambiguous
  - operator association
  - operator precedence