UDC 378.147.34

MODULE «COMPILERS» OF THE COURSE «PROFESSIONAL ENGLISH» FOR BACHELOR DEGREE STUDENTS

Yu.B. Burkatovskaya

National Research Tomsk Polytechnic University

E-mail: tracey@tpu.ru

This paper presents the module «Compilers» as a part of the course «Professional English» for bachelor degree students of the area «Information Systems and Technologies». The module contains 36 hours for seminars, which is a quarter of the total course. The paper describes the seminar devoted to the second stage of compilation – syntax analysis (or parsing).

Key words: professional English, seminar.

1 Overview of the module

The module «Compilers» is a part of the course «Professional English» for bachelor degree students of the area «Information Systems and Technologies». It is delivered in the winter semester for fourth-year students. The module contains 36 hours for seminars, where students study theoretical aspects of compilation, perform individual assessments and projects. Projects include implementing algorithms in a programming language, delivering seminars, presenting theoretical material. Any other options can be considered. At the end of the study students take a credit test.

Course Objectives

1. To learn basics on the formal languages theory necessary to develop a compiler.
2. To understand the structure of a compiler.
3. To know objectives and principles of the compilation stages.

Learning Outcomes

1. Knowledge of the structure of a compiler and of the compilation stages.
2. Knowledge of formal language theory and its application in compilers.
3. Skills in lexical, syntax and semantic analysis of programming languages.
4. Skills in both oral and written scientific communications.

Syllabus

1. Lexical Analysis.
2. Parsing.

3. Semantic Analysis.
4. Optimization.
5. Code Generation.

2 Structure of the seminar

2.1. Lexical analysis and its connection with parsing

Objective: to get a feedback from students. To revise the basics of lexical analysis.

Interaction form: discussion.

Questions to discuss:

1. What is the role of lexical analysis?
2. What a lexer gets as input data?
3. What a lexer provides as output data?
4. What is a token consists of?
5. Which types of tokens are usual for programming languages?
6. What for does lexical analysis use regular expressions?
7. What lexical errors do you know?

2.2. Basics of parsing

Objective: to consider basics of parsing: its input and output data and its objectives and means, necessary to reach the objectives.

Interaction form: frontal teaching.

The second phase of the compiler is *syntax analysis* or *parsing*. The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream. A syntax tree for a given token stream is shown in Fig. 1.

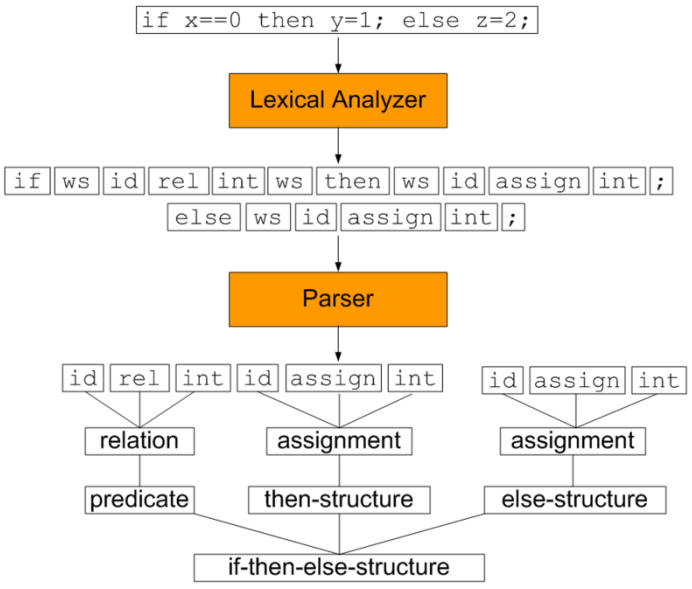

Fig. 1. A syntax tree

Not every string of tokens is a program! By design, every programming language has precise rules that prescribe the syntactic structure of well-formed programs. In our compiler model, the parser obtains a string of tokens from the lexical analyzer and verifies that the string of token names can be generated by the grammar for the source language. We expect the parser to report any syntax errors in an intelligible fashion and to recover from commonly occurring errors to continue processing the remainder of the program. Conceptually, for well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing.

2.3. Difference between lexical and syntax analysis

Objective: to understand a main difference between the structure of valid tokens and valid strings of tokens. To introduce a new mathematical technique for parsing.

Interaction form: discussion.

Question to discuss: Table 1 presents Pascal examples: identifier as a token and if-then-else structure as a string of tokens. What is the principle difference in their structures?

Table 1

Identifier and if-then-else structure

Identifier	**If-then-else structure**
a Ax100s S123456 An_identifier …	if … then … if … then … … else … else …

In Pascal, any identifier is a sequence of letters, digits and '_'-symbols, starting with a letter. It can be described by a regular expression and can be recognized by a finite automata. Any finite automata has the finite memory; consequently, it can count only «mod k». It is sufficient to recognize all possible tokens.

Unlike identifiers and other tokens, the «if-then-else» is a nested structure. It means that one «if-then-else» can include another, end their number is supposed to be unknown and unbounded. Besides, every «if» should have the corresponding «then», and may have the corresponding «else». No finite automata can check this property, because it has to cumulate the number of «if»; so, if the memory size of the finite automata is equal to k,

then it can not handle k+1 structures. To recognize these structures, we need unbounded memory, so-called «stack memory».

2.4. Context-free grammars

Objective: to consider definition of a context-free grammar, an example and the process of derivation.

Interaction form: frontal teaching.

Valid strings of tokens are described by context-free grammars (CFG), and recognized by stack automatons. A CFG consists of terminals, nonterminals, a start symbol, and productions.

1. *Terminals (T)* are the basic symbols from which strings are formed. The term «token name» is a synonym for «terminal» and frequently we will use the word «token» for terminal when it is clear that we are talking about just the token name.
2. *Nonterminals (N)* are syntactic variables that denote sets of strings. Nonterminals impose a hierarchical structure on the language that is key to syntax analysis and translation.
3. In a grammar, one nonterminal is distinguished as the *start symbol (S),* and the set of strings it denotes is the language generated by the grammar.
4. *The productions (P)* of a grammar specify the manner in which the terminals and nonterminals can be combined to form strings. Each *production* consists of:

(a) A nonterminal called the *head* or *left side* of the production; this production defines some of the strings denoted by the head.

(b) Symbol $\rightarrow$.

(c) A *body* or *right side* consisting of zero or more terminals and nonterminals. The components of the body describe one way in which strings of the nonterminal at the head can be constructed.

Consider an example of a grammar for the Pascal «if-then-else» structure.

- EXP → if EXP then ST;
- EXP → if EXP then ST; else ST;
- EXP → **id**
- EXP → **id** COM **id**
- EXP → **id** COM **int**
- ST → **id** = **id** | **id** = **int**
- COM → = | < | <= | > | >=

Let G=<A,T,N,P> be a CFG and:

- $S1S2\ldots Sk\ldots Sn \in (T \cup N \cup \{\varepsilon\})^*$;
- $Sk \rightarrow Y1\ldots Yj \in P$,

then S1S2…Sk…Sn ⇒ S1S2…Y1…Yj …Sn is *a step of derivation*.If there is a sequence α0 ⇒ α1 ⇒ … ⇒ αn, then αn derives from α0 in n steps Consider now an example of derivation:

EXP → if EXP then ST; → if if EXP then ST; else ST; then ST; →…
if if **id** COM **int** then ST; else ST; then **id** = **id**; → …
if if **id** <= **int** then **id** = INT; else **id** = **id**; then **id** = **id**;

2.5. Context-free grammars: assessments

Objective: to develop students' skills in constructing efficient context-free grammars.

Interaction form: individual work.

Assessment: construct context-free grammars for the languages:

- Balanced strings, containing (), [], {};
- Strings in the form 0^n1010^m, n>m;
- Palindromes of odd length, containing symbols 0 and 1;
- Palindromes of even length, containing symbols 0 and 1.

2.6. Trees

Objective: to introduce new terminology connected with trees.

Interaction form: snowball.

Fig. 2 represents an example of a tree with necessary terms. Students are supposed to give definitions for the terms.

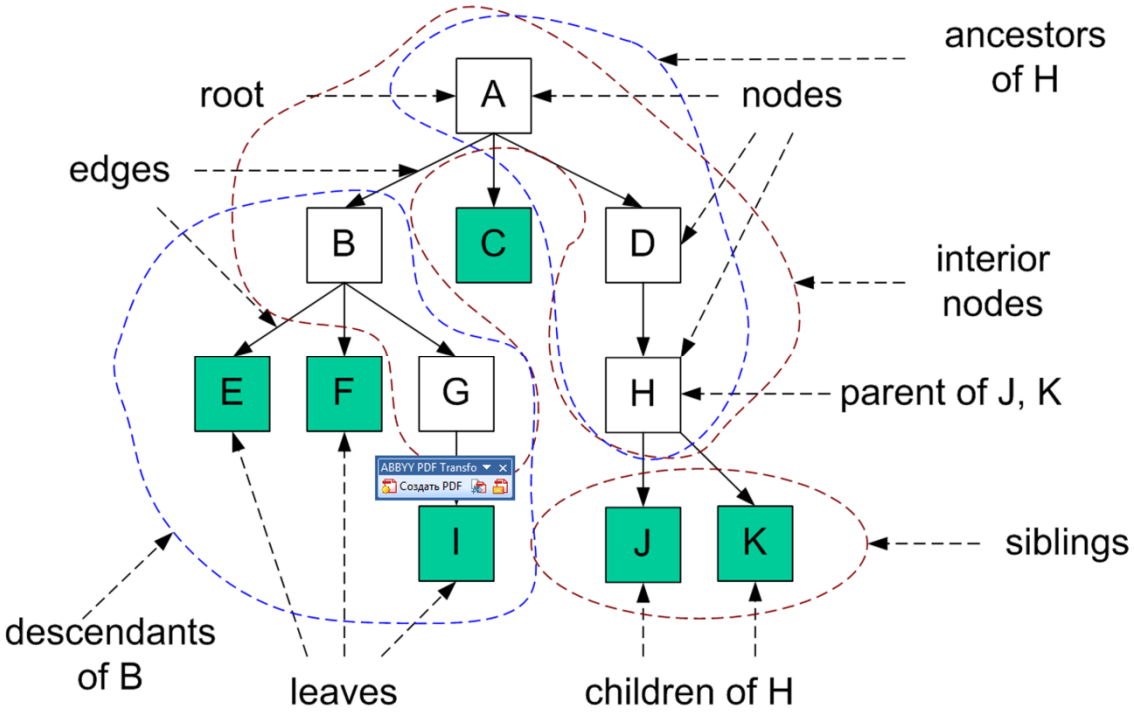

Fig. 2. Tree and corresponding terminology

2.7. Parse tree

Objective: to consider parse trees as one of the main instruments of parsing .

Interaction form: frontal teaching.

A parse tree is a graphical representation of a derivation that filters out the order in which productions are applied to replace nonterminals. Each interior node of a parse tree represents the application of a production. For example, the parse tree in Fig. 4 results from the derivation EXP → if EXP then ST; → if EXP then ID = ID. The leaves of a parse tree are labeled by nonterminals or terminals and, read from left to right, form the *yield* or *frontier* of the tree (marked by the red line).

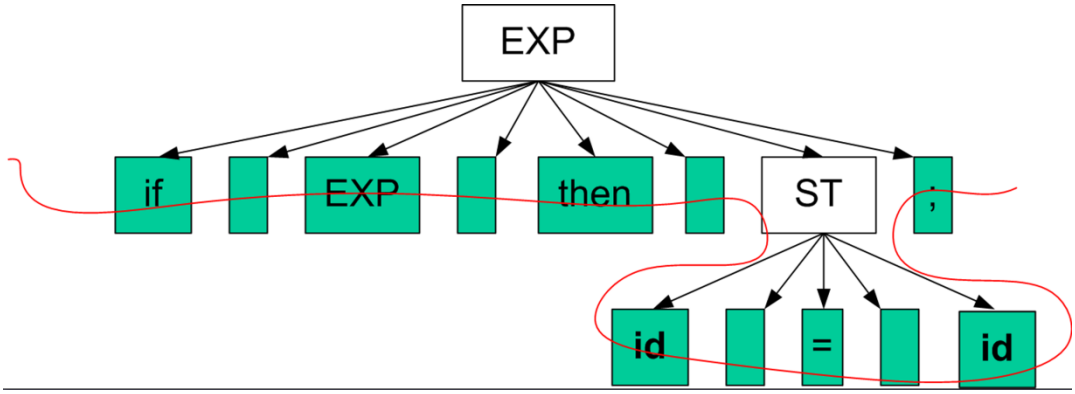

Fig. 3. Parse tree

3. Home assessment

1. To learn new terms.
2. To finish constructing a context-free grammar for arithmetic expressions.
3. To give examples of derivation and the corresponding parse trees.
4. To revise theoretical material.

References

1. AHO, Alfred V., Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. (2006), Compilers: Principles, Techniques, and Tools. Second edition. Peasron. Addison Wesley.
2. HOPCROFT, John E.; Motwani, Rajeev; Ullman, Jeffrey D. (2013). Introduction to Automata Theory, Languages, and Computation (3rd ed.).
3. Compiler design tutorial [Electronic resource]. URL: http://www.tutorialspoint.com/compiler_design/index.htm (retrieved 17.03.2017).
4. Data structures [Electronic resource]. URL: http://btechsmartclass.com/DS/U3_T1.html (retrieved 17.03.2017).