Parsing of General Expressions and Describing Programming Language Syntax using Expressions

Aleksandr Levin
Supersivor: Ing. Stepan Plachy

Department of Theoretical Computer Science
Faculty of Information Technology
Czech Technical University in Prague

June 26, 2024

Aims of the thesis

- Implement LL(1) parser for general expressions.
- Explore a hypothesis that the entire syntax of a programming language can be defined in a form of operators priority table.
- In case of positive result, create an operator priority table for some language (Pascal) as a proof.
- Discuss possible ways of future development of the topic.

Motivation

- **Theoretical interest**: To figure out, whether it is possible to describe language syntax using only expressions.
- **Educational purpose**: Thesis can work as a potential tool for language syntax exploration and testing.
- **Generalization of expressions as a concept**: Expressions, despite being a powerful concept, are mostly treated only as a format for describing different arithmetical and logical operations. In case of positive hypothesis result, it can lead to more generalized utilization of expressions.

General expressions definition

- **Expressions** are defined by a certain set of operators, divided into different priority levels, together with some navigation rules to move across the table.
- To define an **operator**, we must describe it's properties:
  - Arity
  - Associativity
  - Precedence (also known as priority)
  - Left/Right

Standard arithmetic operators priority table

Operators	Associativity	Priority Level
_ + _ _ - _	LEFT	0
_ * _ _ / _	LEFT	1
num (_)	NONE	2

Problem of parsing of general expressions

- **Description**: Given properly configured operators priority table, program must decide, whether the input sentence can be constructed from operators that are defined in the table.
- **Input**: Operators priority table and a sentence to analyze.
- **Output**: Yes or No answer to the question. In the positive case, AST representation of the input sentence.

Solution

- Recursive Descent parsing algorithm that creates a grammar on the go.
- Usage of Parsed Storage structure.
  - Structurally similar to general expressions format.
  - Used to store partially parsed part of input.
  - Content is then used to navigate the parsing process (matching of content to the structure of an operator currently being parsed).

Parsed Storage inner structure

$std :: vector < std :: variant < ParsedStorage, Token >>$

Encountered difficulties and solutions

- **Left recursion** $\rightarrow$ Runtime transformation of left-recursive rules into right-recursive ones.
- **Prefix collision** $\rightarrow$ Parsed storage stores parsed prefix and in case of rule mismatch, it transfers already parsed part to the next rule.
- **Two rules with one being the prefix of another** $\rightarrow$ Parsing order inside the table (from left to right, from top to bottom) in combination with clever rule ordering.

Operators	Associativity	Priority Level
if __ $then$ __ $else$ __ if __ $then$ __	RIGHT	N

Thesis results

- Parser for general expressions is implemented.
- The idea of parsing a programming language using only general expressions was investigated with Pascal language as an example.
- Purely syntactic solution resulted in creation of a language super-set due to the lack of control.
- However, adding some extra functional in the form of type checking should fix this issue.

Operators priority table for Pascal programming language

Operators	Associativity	Priority Level
$program\ \_\_$	NONE	0
$\_\_\ ,\ \_\_$ $\_\_\ ;\ \_\_$ $\_\_\ :\ \_\_$	RIGHT	1
$label\ \_\_$ $const\ \_\_$ $type\ \_\_$ $var\ \_\_$ $procedure\ \_\_$ $function\ \_\_$	NONE	2
$with\ \_\_\ do\ \_\_$ $for\ \_\_\ do\ \_\_$ $repeat\ \_\_\ until\ \_\_$ $while\ \_\_\ do\ \_\_$ $if\ \_\_\ then\ \_\_\ else\ \_\_$ $if\ \_\_\ then\ \_\_$ $goto\ \_\_$	RIGHT	3
$begin\ \_\_\ end$ $case\ \_\_$ of end $case\ \_\_\ of\ \_\_\ end$	NONE	4

...

Example of a purely syntactic solution flaw

```
program ImperfectionTest;
begin
  program HelloWorld;
  begin
    writeln('Hello, world!');
  end;
end;
```

- Program definition must be unique, however there is no way to apply this restriction while using general expressions format.
- Still, if we would include some **program definition** data type, this issue should be fixed.

Thank you for your attention!

Aleksandr Levin
Supersivor: Ing. Stepan Plachy
levinale@fit.cvut.cz

Question 1

Is there any literature on the same topics you were trying to solve? I.e., the general parsers for expressions and the possibility of expressing the whole language using the expressions?

- In general, of course there are some mentions of expressions in different books, related to parsing (Compilers, ISBN 0-321-48681-1 as an example).
- However, expressions there are mentioned only as a format for arithmetic, logic and other math-related elements description, not as a thing that is capable of parsing programming languages.
- About general expressions parsers, I haven't managed to find any literature.

Question 2

Did you consider other languages than Pascal? Could they be expressed using general expressions parser?

- Main languages considered were: C/C++, Lua and Pascal.
  - **C/C++** had some nicely-defined grammars (from Microsoft for example), however these languages are somewhat low level and quiet technical, so they were hard to understand for a potential reader.
  - **Lua** had small enough and well-organised official grammar, but it had some elements which were complicating grammar translation into general expression format (namely, optional tokens and elements with optional cycling).
  - **Pascal** provided official grammar with well-defined language syntax, which also was easy to translate into general expressions format, so I picked Pascal as an example language.
- Highly likely, yes, almost half of Lua language grammar was translated before it became too complex and C/C++ grammar was structurally simple, it was just hard to understand all of it's elements.