Compiler JYTHON to Python

Team : Senior AirKickers
Members :
1. Ysmagul Aisara
2. Bakytzhankyzy Zarina
3. Mussin Nurlybek
4. Aspandiyar Shansharkhan
5. Orazkulov Sayat

Introduction

During this lesson we went from basics to making our own computer. In this case, we will work on a project that will compile code from one language to another. The main aim of the project is to understand the principles which are used in working of compilers, make attempt to create new language with its converter to python and develop programming skills. This particular project will translate simple expressions, if-statements and loops.

A compiler reads the source code of a program, and produces either machine code (instructions in a suitable format for the CPU to execute), bytecode, or a translation of the source into some other form. In our case we decided to take a JYTHON code, which is a mixture if Java and Python and get as an output Python code.
So we divided our task into 4 main steps:
- Tokenizer
- Parser
- Emitter
- Compiler

Lexical Analyzer (Tokenizer)

Lexical analysis is the process of converting a sequence of characters (such as in a computer program or web page) into a sequence of tokens (strings with an identified "meaning"). A program that performs lexical analysis may be called a lexer, tokenizer, or scanner (though "scanner" is also used to refer to the first stage of a lexer).

Task on this moment of program was to create a lexical analyzer for the simple programming language specified below. The program should read input from a file and/or stdin, and write output to a file and/or stdout. If the language being used has a lexer module/library/class, it would be great if two versions of the solution are provided: One without the lexer module, and one with.

Input Specification

Operators:

Name	Common name	Character sequence
`Op_multiply`	multiply	`*`
`Op_divide`	divide	`/`
`Op_mod`	mod	`%`
`Op_add`	plus	`+`
`Op_subtract`	minus	`-`
`Op_negate`	unary minus	`-`
`Op_less`	less than	`<`
`Op_lessequal`	less than or equal	`<=`
`Op_greater`	greater than	`>`
`Op_greaterequal`	greater than or equal	`>=`
`Op_equal`	equal	`==`
`Op_notequal`	not equal	`!=`
`Op_not`	unary not	`!`
`Op_assign`	assignment	`=`
`Op_and`	logical and	`&&`
`Op_or`	logical or	`||`

Symbols:

Name	Common name	Character
`LeftParen`	left parenthesis	`(`
`RightParen`	right parenthesis	`)`
`LeftBrace`	left brace	`{`
`RightBrace`	right brace	`}`
`Semicolon`	semi-colon	`;`
`Comma`	comma	`,`

Keywords:

Name	Character sequence
`Keyword_if`	`if`
`Keyword_else`	`else`
`Keyword_while`	`while`
`Keyword_print`	`print`
`Keyword_putc`	`putc`

Identifiers and literals:

Name	Common name	Format description	Format regex	Value
`Identifier`	identifier	one or more letter/number/ underscore characters, but not starting with a number	`[_a-zA-Z][_a-zA-Z0-9]*`	as is
`Integer`	integer literal	one or more digits	`[0-9]+`	as is, interpreted as a number
`Integer`	char literal	exactly one character (anything except newline or single quote) or one of the allowed escape sequences, enclosed by single quotes	`'([^'\n]|\\n|\\\\)'`	the ASCII code point number of the character, e.g. 65 for `'A'` and 10 for `'\n'`
`String`	string literal	zero or more characters (anything except newline or double quote), enclosed by double quotes	`"[^"\n]*"`	the characters without the double quotes and with escape sequences converted

Zero-width tokens:

Name	Location
`End_of_input`	when the end of the input stream is reached

White space:

- Zero or more whitespace characters, or comments enclosed in `/* ... */`, are allowed between any two tokens, with the exceptions noted below.
- "Longest token matching" is used to resolve conflicts (e.g., in order to match **<=** as a single token rather than the two tokens **<** and **=**).
- Whitespace is *required* between two tokens that have an alphanumeric character or underscore at the edge.
  - This means: keywords, identifiers, and integer literals.
  - e.g. `ifprint` is recognized as an identifier, instead of the keywords `if` and `print`.
  - e.g. `42fred` is invalid, and neither recognized as a number nor an identifier.
- Whitespace is *not allowed* inside of tokens (except for chars and strings where they are part of the value).
  - e.g. `& &` is invalid, and not interpreted as the `&&` operator.

Complete list of token names

```
End_of_input  Op_multiply   Op_divide     Op_mod        Op_add
Op_subtract   Op_negate     Op_not        Op_less
Op_lessequal  Op_greater    Op_greaterequal             Op_equal
Op_notequal   Op_assign     Op_and        Op_or        Keyword_if
Keyword_else  Keyword_while Keyword_print Keyword_putc LeftParen
RightParen.   LeftBrace     RightBrace    Semicolon     Comma
Identifier    Integer       String
```

Output Format：

The program output should be a sequence of lines, each consisting of the following whitespace-separated fields:

1. the line number where the token starts
2. the column number where the token starts
3. the token name
4. the token value (only for `Identifier`, `Integer`, and `String` tokens)
5. the number of spaces between fields is up to you. Neatly aligned is nice, but not a requirement.

Syntax Analyzer

A Syntax analyzer transforms a token stream (from the Lexical analyzer) into a Syntax tree, based on a grammar.

Task was to take the output from the Lexical analyzer task, and convert it to an Abstract Syntax Tree (AST), based on the grammar below. The output should be in a flattened format. The resulting AST should be formulated as a Binary Tree.

The following table shows the input to lex, lex output, and the AST produced by the parser:

Input to lex	Output from lex, input to parse	Output from parse

count = 1;
while (count < 6) {
  if (count % 2 == 0){
  print(count);
  print("\n");
  }
  count = count + 1;
}
	1 1 Identifier count
	1 7 Op_assign
	1 9 Integer 1
	1 10 Semicolon
	2 1 Keyword_while
	2 7 LeftParen
	2 8 Identifier count
	2 14 Op_less
	2 16 Integer 6
	2 17 RightParen
	2 19 LeftBrace
	3 5 Keyword_if
	3 8 LeftParen
	3 9 Identifier count
	3 15 Op_mod
	3 17 Integer 2
	3 19 Op_equal
	3 22 Integer 0
	3 23 RightParen
	3 24 LeftBrace
	4 9 Keyword_print
	4 14 LeftParen
	4 15 Identifier count
	4 20 RightParen
	4 21 Semicolon
	5 9 Keyword_print
	5 14 LeftParen
	5 15 String "\n"
	5 19 RightParen
	5 20 Semicolon
	6 5 RightBrace
	7 5 Identifier count
	7 11 Op_assign
	7 13 Identifier count
	7 19 Op_add
	7 21 Integer 1
	7 22 Semicolon
	8 1 RightBrace
	9 1 End_of_input
		Sequence
		Sequence
		;
		Assign
		Identifier count
		Integer 1
		While
		Less
		Identifier count
		Integer 6
		Sequence
		Sequence
		;
		If
		Equal
		Mod
		Identifier count
		Integer 2
		Integer 0
		If
		Sequence
		Sequence
		;
		Sequence
		;
		Prti
		Identifier count
		;
		Sequence
		;
		Prts
		String "\n"
		;
		;
		Assign
		Identifier count
		Add
		Identifier count
		Integer 1

To implement our compiler, we need to build an AST. In computer science, an abstract syntax tree (AST), or just syntax tree, is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code. The syntax is "abstract" in the sense that it does not represent every detail appearing in the real syntax, but rather just the structural, content-related details. For instance, grouping parentheses are implicit in the tree structure, and a syntactic construct like an if-condition-then expression may be denoted by means of a single node with three branches.

This distinguishes abstract syntax trees from concrete syntax trees, traditionally designated parse trees, which are typically built by a parser during the source code translation and compiling process. Once built, additional information is added to the AST by means of subsequent processing, e.g., contextual analysis.

The design of an AST is often closely linked with the design of a compiler and its expected features.
Core requirements include the following:

- Variable types must be preserved, as well as the location of each declaration in source code.
- The order of executable statements must be explicitly represented and well defined.
- Left and right components of binary operations must be stored and correctly identified.
- Identifiers and their assigned values must be stored for assignment statements.

Some operations will always require two elements, such as the two terms for addition. However, some language constructs require an arbitrarily large number of children, such as argument lists passed to programs from the command shell. As a result, an AST used to represent code written in such a language has to also be flexible enough to allow for quick addition of an unknown quantity of children.

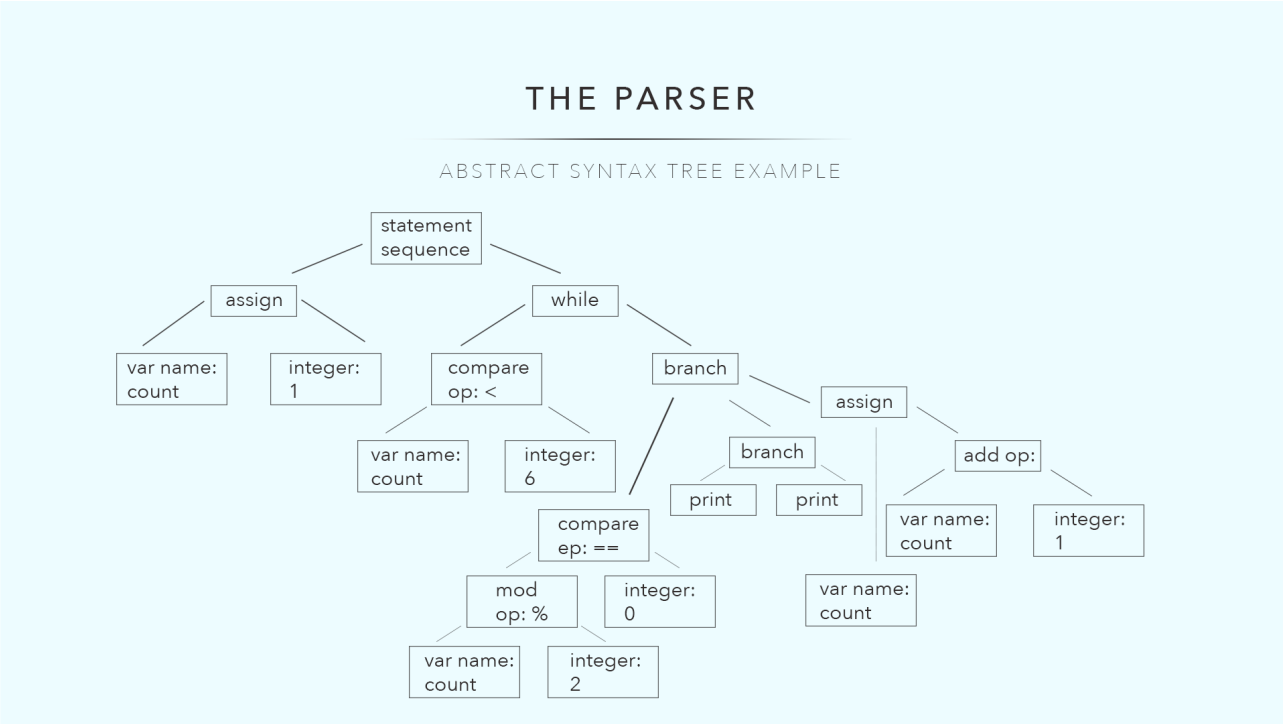
Another major design requirement for an AST is that it should be possible to unparse an AST into source code form.The source code produced should be sufficiently similar to the original in appearance and identical in execution, upon recompilation.

For our code:

```
count = 1;

while (count < 6) {
  if (count % 2 == 0){
     print(count);
     print("\n");
  }
  count = count + 1;
}
```

we will get as an output AST tree like:



Conclusion

A language rewriter is usually a program that translates the form of expressions without a change of language. The term compiler-compiler refers to tools used to create parsers that perform syntax analysis.

A compiler is likely to perform many or all of the following operations: preprocessing, lexical analysis, parsing, semantic analysis (syntax-directed translation), conversion of input programs to an intermediate representation, code optimization and code generation. Compilers implement these operations in phases that promote efficient design and correct transformations of source input to target output. Program faults caused by incorrect compiler behavior can be very difficult to track down and work around; therefore, compiler implementers invest significant effort to ensure compiler correctness.

Compilers are not the only language processor used to transform source programs. An interpreter is computer software that transforms and then executes the indicated operations. The translation process influences the design of computer languages which leads to a preference of compilation or interpretation. In practice, an interpreter can be implemented for compiled languages and compilers can be implemented for interpreted languages.

For the source code we used our own newly created language. For generating compiler front- and back- end modules were performed. As result we got translated python code. The syntax tree is designed by using the predefined Grammar of the language and the input code. Since most of the programming languages use Context-Free Grammar, in this project it was also used as a basic grammar. Because of simple nature of source code, we did not include Semantic Analyzing.

References

https://rosettacode.org/wiki/Compiler
https://en.wikipedia.org/wiki/Lexical_analysis
https://en.wikipedia.org/wiki/Compiler
https://www.geeksforgeeks.org/compiler-design-introduction-to-syntax-analysis/
https://www.geeksforgeeks.org/compiler-design-phases-compiler/
https://en.wikipedia.org/wiki/Abstract_syntax_tree
https://www.tutorialspoint.com/compiler_design/compiler_design_lexical_analysis.htm