Started on	Friday, 6 May 2022, 5:16 PM
State	Finished
Completed on	Thursday, 19 May 2022, 8:24 PM
Time taken	13 days 3 hours
Marks	30.25/34.00
Grade	**88.97** out of 100.00

Information

In the compiler super-quiz you will implement parts of a compiler for a small programming language described below and discussed in the lectures. It comprises three quizzes about

1. the scanner,
2. the parser and
3. the code generator.

You will get most of the marks for implementing the compiler and a small part for additional questions about compilers.

The implementation parts of the quizzes are incremental: the parser requires a working scanner and the code generator requires a working parser. You will receive Python 3 templates for each of the three parts. You have to study these programs to understand how they work and you will be asked to extend them.

Feedback about the errors is limited. It is therefore essential that you thoroughly test your programs before you submit them (see below).

Templates and Precheck

Code that you submit to a question must be based on the template given in that question. The precheck goes some way to ensure this, but this remains your own responsibility. Please use the Precheck button before submitting your answer using the Check button. A precheck attracts no marks and no penalty; you can use it any number of times and also after incorrect submissions. If your code fails the precheck, it will fail the full check as well. If it passes the precheck, it may or may not pass the full check. Note that a precheck will not do any testing of your code; even syntactically incorrect submissions based on the provided template may pass it.

Your code should be derived from the given template by modifying only the parts mentioned in the requirements; other changes are at your own risk. Feel free, however, to experiment during the development.

Syntax

The compiler will accept programs with the following syntax:

Program = *Statements*
Statements = *Statement* (`;` *Statement*)*
Statement = *If* | *While* | *Assignment*

If = `if` *Comparison* `then` *Statements* `end`
While = `while` *Comparison* `do` *Statements* `end`
Assignment = `identifier` `:=` *Expression*

Comparison = *Expression Relation Expression*
Relation = `=` | `!=` | `<` | `<=` | `>` | `>=`

Expression = *Term* ((`+` | `-`) *Term*)*
Term = *Factor* ((`*` | `/`) *Factor*)*
Factor = (*Expression*) | `number` | `identifier`

Identifiers contain only lower-case letters. Numbers are represented by non-negative integers in base-10 notation. In the following quizzes you will need to modify this BNF for if-then-else-end statements, write statements, read statements and Boolean expressions. An example of a program using the above BNF extended by read and write statements is:

```
read n;
sum := 0;
while n > 0 do
  sum := sum + n;
  n := n - 1
end;
write sum
```

Testing

You will need to create your own programs for thoroughly testing your compiler.

This means you will need to create a comprehensive collection of test inputs with the aim of covering all parts of the specification and especially any parts of the code you have changed.

If your submission fails for any input, the code certainly contains at least one error. You will need to identify the error and correct it before resubmitting, not just trying some changes to the code. You should not stop looking for errors when you find the first one; there may be others.

Note that testing provides no guarantees. Your program might work for all your tests and still fail some or even most tests on the quiz server. You will need to create tests with a good coverage.

Some methods for finding errors are:

- read and understand all parts of the specification (including this information box);

- understand what each part of the program is supposed to do;
- create test cases covering all parts of the code that have been changed;
- systematically create sequences of characters (possible inputs);
- randomly create possible inputs;
- carefully review parts of the code that have been changed;
- check if all variables are assigned to before they are used;
- check if every expression is defined in all cases: if it is not, try to come up with an input that causes it to be undefined;
- check if all loops and recursions terminate;
- check for every piece of the code: why is it there? can it fail? if so, under which circumstances? is there an input that causes these circumstances?

Testing works best as a combination of being systematic and creative; try to come up with unusual inputs.

Information

By submitting your solution below you confirm that it is entirely your own work.

Your submissions are logged and originality detection software will be used to compare your solution with other solutions. Dishonest practice, which includes

- letting someone else create all or part of an item of work,
- copying all or part of an item of work from another person with or without modification, and
- allowing someone else to copy all or part of an item of work,

may lead to partial or total loss of marks, no grade being awarded and other serious consequences including notification of the University Proctor.

You are encouraged to discuss the general aspects of a problem with others. However, anything you submit for credit must be entirely your own work and not copied, with or without modification, from any other person. If you need help with specific details relating to your work, or are not sure what you are allowed to do, contact your tutors or lecturer for advice. If you copy someone else's work or share details of your work with anybody else, you are likely to be in breach of university regulations and the Computer Science and Software Engineering department's policy. For further information please see:

- Academic Integrity Guidance for Staff and Students
- Academic Misconduct Regulations

Question **1**

Partially correct

Mark 0.25 out of 1.00

Extend the BNF in the answer box below to include:

1. if-then-else-end statements (in addition to the if-then statements it already includes)
2. read statements of the form `read` *i* that reads a value and stores it in the variable with identifier *i*
3. write statements of the form `write` *e* that print the value of the expression *e*

Notes:

- Tokens must be written in lowercase letters, e.g. `if`, `identifier`. (The provided BNF shows non-terminals in a `monospaced` font, but you do not need to worry about monospacing for your answers.)
- Non-terminals must start with a capital letter, e.g. *If*, *Expression*. (The provided BNF shows non-terminals in *italics*, but you do not need to worry about italics for your answers.)

Answer: (penalty regime: 10, 20, ... %)

Program = *Statements*
Statements = *Statement* (`;` *Statement*)*
Statement = *If* | *While* | *Assignment* |Read|Write
If = `if` *Comparison* `then` *Statements* [else Statements] `end`
While = `while` *Comparison* `do` *Statements* `end`
Assignment = `identifier` `:=` *Expression*
Read = i
Write = e

Comparison = *Expression Relation Expression*
Relation = `=` | `!=` | `<` | `<=` | `>` | `>=`

Expression = *Term* ((`+` | `-`) *Term*)*
Term = *Factor* ((`*` | `/`) *Factor*)*
Factor = `(`*Expression*`)` | `number` | `identifier`

Non-terminal	Got	
`Statement`	`|Read|Write`	✔
`If`	`[else Statements]`	✔
`Read`	`i`	✖
`Write`	`e`	✖

Partially correct

Marks for this submission: 0.50/1.00. Accounting for previous tries, this gives **0.25/1.00**.

Question **2**

Correct

Mark 20.00 out of 20.00

Please download the parser template.

1. Copy the changes to the scanner template you made in the previous quiz into the parser template. Note that the scanner is now called from the parser, which is different to how it was tested in the previous quiz. This might uncover errors in your scanner which went unnoticed so far; you will need to fix such errors.
2. Read and understand the parser template. In particular, you need to understand how it represents the abstract syntax tree (AST), how the recursive-descent parser uses the scanner, how it constructs the AST and how the `indented` methods of the AST classes work.
3. The classes `Program_AST`, ..., `Identifier_AST` represent the AST. Read some of these classes and understand the purpose of the methods `__init__`, `__repr__` and `indented`. Introduce a new class `If_Else_AST` for if-then-else-end statements and implement the methods `__init__`, `__repr__` and `indented`. The class will be similar to `If_AST`. The method `If_Else_AST.indented` should output the label `If-Else` followed by the condition, the then-part and the else-part.
4. The methods `program`, ..., `identifier` implement the recursive-descent parser. Read and understand this part of the code. Extend the method `if_statement` so that it can parse both if-then-end and if-then-else-end statements. The method should construct the appropriate syntax tree in each case.
5. Extend the recursive-descent parser by the new statement `write` *e* that prints the value of the expression *e*. To this end, modify the method `statement` and introduce a new method `write` which constructs the syntax tree using `Write_AST`.
6. Extend the recursive-descent parser by the new statement `read` *i* that reads a value and stores it in the variable with identifier *i*. To this end, modify the method `statement` and introduce a new method `read` which constructs the syntax tree using `Read_AST`.

The parser is called to show the AST of the input, which is a program. For example, assume the parser is in the file `parser.py` and the file `program` contains

```
z := 2;
if z < 3 then
  z := 1
else
  z := 0
end
```

Running the parser by `python3 parser.py < program > tree` will produce the file `tree` which contains

```
Statements
    Assign
        z
        2
    If-Else
        <
            z
            3
        Statements
            Assign
                z
                1
        Statements
            Assign
                z
                0
```

This preorder traversal of the AST indicates the levels by indentation.

On submission your code is tested with several programs. Tests are carried out using the following procedure, which you should reproduce using your own programs.

1. Your parser is run using Python 3 with the program passed via standard input. The AST is expected via standard output. Assuming your parser is in the file `parser.py`, the program is in the file `program`, and the AST goes to the file `tree`, this amounts to calling `python3 parser.py < program > tree`

An error at this stage is indicated by
`Error in compiling X (HINT)`
`X` is an identifier of the program and `HINT` is a brief description of the constructs it uses.

2. If stage 1 was successful, the AST is compared with the expected AST. A difference is indicated by
`Wrong output for X (HINT)`
3. If there is no difference, you pass this test.

The final line of feedback shows the total number of tests that your parser has passed. Each test counts equally towards the marks for this question.

Your first submission to this question will attract no penalty. Thereafter each wrong submission attracts 10% penalty.

Answer: (penalty regime: 0, 10, ... %)

```
import re
import sys

class Scanner:
    '''The interface comprises the methods lookahead and consume.
       Other methods should not be called from outside of this class.'''

    def __init__(self, input_file):
        '''Reads the whole input_file to input_string, which remains constant.
           current_char_index counts how many characters of input_string have
           been consumed.
           current_token holds the most recently found token and the
           corresponding part of input_string.'''
        # source code of the program to be compiled
        self.input_string = input_file.read()
        # index where the unprocessed part of input_string starts
        self.current_char_index = 0
        # a pair (most recently read token, matched substring of input_string)
        self.current_token = self.get_token()

    def skip_white_space(self):
```

	Got
✔	Correct output for program 05 (write, addition, subtraction, assignments) Correct output for program 06 (write, while-do-end, if-then-end, addition, assignments) Correct output for program 07 (read, while-do-end, if-then-end, addition, division, assignments) Correct output for program 08 (read, write, while-do-end, addition, subtraction, assignments) Correct output for program 09 (read, write, nested while-do-end, multiplication, subtraction, assignments) Correct output for program 10 (if-then-else-end, division, assignments) Correct output for program 11 (read, write, nested while-do-end, nested if-then-else-end, inequality, subtraction, assignments) Correct output for program 14 (write, while-do-end, if-then-end, nested if-then-else-end, arithmetic, assignments) Correct output for program 15 (read, write, nested while-do-end, if-then-end, if-then-else-end, arithmetic, assignments) Correct output for program 17 (while-do-end, subtraction, assignments, lexical error) Correct output for program 18 (assignments, syntax error) 11 of 11 tests correct

Passed all tests! ✔

Correct

Marks for this submission: 20.00/20.00.

Question **3**

Correct

Mark 7.00 out of 10.00

In this question you will create a parser for the same language using the parser generator PLY.

Please download the PLY parser template.

1. Copy the changes to the PLY scanner template you made in the previous quiz into the PLY parser template.
2. Read and understand the PLY parser template. Note that it is only necessary to specify the grammar rules and corresponding functions that construct syntax trees. PLY generates a bottom-up (LALR) parser from these rules, which works differently from a top-down (recursive descent) parser. Each function is called after the right-hand side of the corresponding rule has been parsed, whereas in a recursive descent parser the functions are called before the right-hand side is parsed as they are responsible for parsing.
3. Copy the class `If_Else_AST` for if-then-else-end statements you have developed for the previous parser into the PLY parser template.
4. Extend the PLY parser so that it can parse both if-then-end and if-then-else-end statements, as well as the statements `write` *e* and `read` *i* as described for the previous parser. Construct the appropriate syntax tree in each case.

The parser using PLY can be run in the same way as the previous parser and should generate the same output. On submission, it will be tested in the same way. You should again test with your own programs.

Your first submission to this question will attract no penalty. Thereafter each wrong submission attracts 10% penalty.

Answer: (penalty regime: 0, 10, ... %)

```
"""
This program uses PLY (Python Lex-Yacc). Documentation for PLY is
available at
    https://www.dabeaz.com/ply/ply.html

PLY can be installed on your own system using pip, which comes
preinstalled on recent versions of Python (>= 3.4). Using pip the PLY
package can be installed with the following command:
    pip3 install ply
This requires Internet access to download the package.
"""

import ply.lex as lex
import ply.yacc as yacc
import sys

# reserved words
reserved = {
    'do': 'DO',
    'else': 'ELSE',
    'read': 'READ',
```

	Got
✔	Correct output for program 05 (write, addition, subtraction, assignments) Correct output for program 06 (write, while-do-end, if-then-end, addition, assignments) Correct output for program 07 (read, while-do-end, if-then-end, addition, division, assignments) Correct output for program 08 (read, write, while-do-end, addition, subtraction, assignments) Correct output for program 09 (read, write, nested while-do-end, multiplication, subtraction, assignments) Correct output for program 10 (if-then-else-end, division, assignments) Correct output for program 11 (read, write, nested while-do-end, nested if-then-else-end, inequality, subtraction, assignments) Correct output for program 14 (write, while-do-end, if-then-end, nested if-then-else-end, arithmetic, assignments) Correct output for program 15 (read, write, nested while-do-end, if-then-end, if-then-else-end, arithmetic, assignments) Correct output for program 17 (while-do-end, subtraction, assignments, lexical error) Correct output for program 18 (assignments, syntax error) 11 of 11 tests correct

Passed all tests! ✔

Correct

Marks for this submission: 10.00/10.00. Accounting for previous tries, this gives **7.00/10.00**.

Question **4**

Correct

Mark 1.00 out of 1.00

How many syntax trees does the string `1+2*3+4` have according to the following BNF?

Expression = Expression Arithmetic Expression | `number`
Arithmetic = + | – | * | /

Answer: 5

Correct

Marks for this submission: 1.00/1.00.

Question **5**

Correct

Mark 1.00 out of 1.00

How many syntax trees does the string `1+2*3+4` have according to the following BNF?

Expression = Term | Expression Additive Term
Additive = + | –
Term = Factor | Term Multiplicative Factor
Multiplicative = * | /
Factor = (Expression) | `number`

Answer: 1

Correct

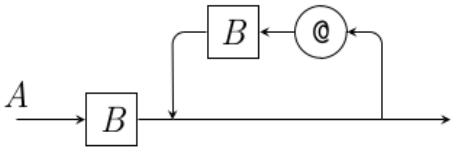
Marks for this submission: 1.00/1.00.

Question **6**

Correct

Mark 1.00 out of 1.00

Which extended-BNF rule corresponds to the following syntax diagram?



Select one:

- A B | @ B
- A = B (B @)*
- A = B (@ B)*
- A = B | B @
- A B (@ B)*
- A (B @)* B
- A B (B @)*
- A = (B @)* B
- A = B | @ B
- A B | B @

Your answer is correct.

Correct

Marks for this submission: 1.00/1.00.

◄ Quiz 7 = Assignment Part 1: Lexical Analysis

Jump to...

Quiz 9 = Assignment Part 3: Code Generation ►