TKOM2021 - C++ do Python

Konrad Kulesza 300247

Podzbiór języka wejściowego

- typy danych:
  - int
  - std::string
  - bool
- komentarze jedno- i wielolinijkowe
- zmienne lokalne
- operacje arytmetyczne
- wypisanie na ekran
  - z oraz bez znaku nowej linii na końcu
- wywołania funkcji
- instrukcje złożone:
  - if-else
  - while
- deklarowanie funkcji
  - argumenty
  - wartość zwracana

`W porównaniu do sprawozdania wstępnego zrezygnowałem z typu danych "float" oraz wyrażenia "for" ze względu sugestię Pani prowadzącej oraz wtórność rozwiązań`

Założenia

- w kodzie dopuszczone są tylko znaki alfanumeryczne - w stringach dowolne
- wyrażenia zaczynające się znakiem `#` na przykład `#include<iostream>` są przezroczyste, lekser nie przekazuje ich do dalszej analizy
- jeden plik na wejściu
- jeden plik na wyjściu
- kod wejściowy zgodny ze standardem języka C++11
- kod wynikowy zgodny ze standardem języka Python 3.0
- dopuszczalne(ale komunikowane użytkownikowi jako ostrzeżenia) są błędy w kodzie wejściowym, które mimo tego że nie pozwalają na skompilowanie programu wejściowego to są dopuszczalne w kodzie wynikowym:
  - nadpisywanie słów kluczowych języka wejściowego.
  - przypisanie wartości do zmiennej, która nie została wcześniej zadeklarowana - ponieważ w języki Python przypisanie wartości do zmiennej nie różni się od deklaracji.
- nadpisywanie słów kluczowych języka wyjściowego skutkuje przerwaniem przetwarzania, ponieważ celem tłumaczenia jest wygenerowanie programu w pełni wykonywalnego

- po przetłumaczeniu całego kodu w pliku wynikowym zostaje dodana instrukcja:

```
if __name__ == '__main__':
    main()
```

  w celu możliwości wykonania danego skryptu w taki sam sposób jak plik wejściowy

Gramatyka

```
program             = [<preprocesor_macro>] <instruction_block> <end_of_file>

### #include<iostream>....
preprocesor_macro   = "#" {any_char_without_nl}  <end_of_line>

### komentarze
comment             = <single_line_comment> | <multi_line_comment>
multi_line_comment  = "/*" <string_char> "*/"
single_line_comment = "//" <string_char> <end_of_line>

### instrukcje złożone
instruction_block   = {<simple_instruction> | <complex_instruction>}
scope               = "{" <instruction_block> "}"
function_declaration= <type> <identifier> "(" [<type><identifier>[{","<type>
<identifier}]] ")" <scope>
while_statement     = "while" "(" <condition> ")"  <scope>
if_statement        = "if" "(" <condition> ")" <scope> ["else"  <scope>]
complex_instruction = <if_statment> | <while_statement>

### instrukcje proste
variable_declaration= <type> <identifier> [ "=" <right_value>]
variable_assignment = <identifier> "=" <right_value>
print_no_new_line   = <print_with_new_line> "<<" "std::endl"
print_with_new_line = "std::cout" "<<" <right_value>
return_expression   = "return" <r_value>
function_invocation = <identifier> "(" [<right_value> {"," <right_value>}]

simple_instruction  = (<print_no_new_line> | <print_with_new_line> |
<variable_assignment> | <variable_declaration> | <function_invocation> |
return_expression ) <end_of_ins>


### operacje arytmetyczne i warunki
aritmetic_operation = <arithmetic_component> <arithmetic_operand>
<arithmetic_component>
aritmetic_component = ( "("<arithmetic_operation>")" ) | <literal> |
<identifier>| <arithmetic_operation>
right_value         = <literal> | <aritmetic_operation> | <identifier>

condition           = <comparison> | <literal> | <identifier>
comparison          = <comparison_operand> <comparison_operator>
<comparison_operand>
comparison_operand  = <literal> | <identifier>
```

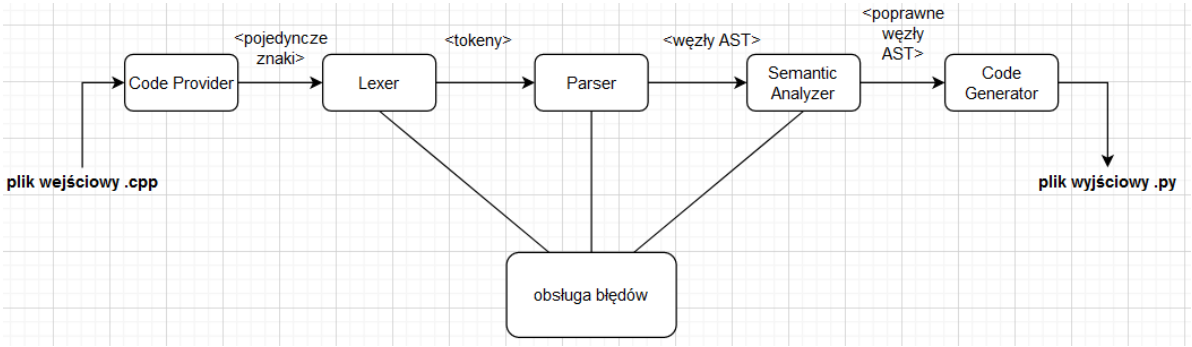
```
### literały, typ i identyfikator
literal             = <bool_literal> | <string_literal> | <integer_literal>
string_literal      = <quote> char_string <quote>
bool_literal        = "true" | "false"
integer_literal     = <non_zero_digit> {digit}
char_string         = <any_char_without_eol> | <enf_of_line>
type                = "bool" | "int" | "std::string"
identifier          = {char}

###operatory
aritmetic_opeator   = "+" | "-" | "*" | "/"
comparison_operator = "!=" | "==" | ">" | "<" | ">=" | "<="

###podstawowe
end_of_file         = EOF
end_of_line         = "\n"
end_of_ins          = ";"
any_char_without_eol= <char> | <special_char> | <quote> | <end_of_ins>
char                = <digit> | <alphabet_char>
alphabet_char       = [a-z] | [A-Z]
special_char        = "!" | "@" | "#" | "%" | "^" | "&" ...
quote               = <double_quote> | <single_quote>
double_quote        = """
single_quote        = "'"
digit               = "0"|<non_zero_digit>
non_zero_digit      = "1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
```

Implementacja

Widok z góry systemu - potok przetwarzania



Opis

Implementacja całego systemu została wykonana w języku Python3 w IDE PyCharm oraz jednego skryptu ułatwiającego sprawdzanie wyniku napisanego w Bashu.

System jest podzielony na moduły, które łącznie tworzą potok przetwarzania. Na obrazku wyżej widać, że błędy w kodzie wejściowym mogą być sygnalizowane na wielu poziomach. Na szczególną uwagę zasługuje `Semantic Analyzer` - jest on,w przypadku poprawnego pliku wejściowego, "przezroczystą" warstwą systemu, która definiuje tablicę symboli, waliduje otrzymane struktury i przepuszcza je bez zmiany. Dzięki niemu, moduł generowania kodu nie zgłasza błędów, ponieważ dostaje na swoje wejście sprawdzone pod każdym względem drzewo

rozbioru, które musi tylko przetłumaczyć na kod wynikowy. Dzięki takiej strukturze projektu, dopisywanie kolejnych funkcjonalności i modyfikowanie przyjętych rozwiązań jest bardzo wygodne, ponieważ programista nie musi skupiać się na całości, a jedynie na fragmencie, którego dotyczy dana zmiana.

Punktowy opis modułów

- Code Provider:
  - Wrapper na strumień wejściowych znaków.
  - Odpowiedzialny za dostarczanie pojedynczych znaków kodu wejściowego do parsera oraz śledzenia miejsca w kodzie.
  - Nie produkuję błędów.
- Lexer:
  - Składa dostarczone znaki w Tokeny.
  - Pomija białe znaki.
  - Produkuję błędy związane z niepoprawnymi tokenami, np. niezakończenie komentarze wielolinjkowego
- Parser:
  - Składa dostarczone Tokeny w węzły drzewa składniowego.
  - Nie pozwala na nadpisywanie słów kluczowych języka wyjściowego.
  - Nie pozwala na tworzenie "pustych struktur", np. w instrukcji `if` musi być przynajmniej jedna instrukcja nie będąca komentarzem.
- Semantic analyzer:
  - Przezroczysty w przypadku poprawnych struktur
  - Przepuszcza poprawne węzły AST
  - Konstruuję tablicę symboli oraz ją analizuje.
  - Nie pozwala na m.in. przypisanie do zmiennej złego typu wyrażenia.
- Code Generator:
  - Tłumaczy otrzymane na wejściu drzewa AST na kod w języku Python.
  - Uwzględnia odpowiednie wcięcia w zależności od poziomu zagnieżdżenia instrukcji.
  - Nie produkuję błędów

Struktury danych

- Token - przechowuje: typ tokenu, wartość, pozycja w kodzie wejściowym
- Symbole
  - zmiennych - nazwa oraz typ
  - funkcji - nazwa, typ wartości zwracanej, typy argumentów wejściowych
- AstNode - klasa bazowa węzła drzewa rozbioru, każdy element drzewa dziedziczy po niej
  - VariableAssignment, FunctionDeclaration itd...
- Błędy - opisane niżej.

Obsługa błędów

Każdy z "środkowych" modułów systemu ma swoją rodzinę błędów.

Przykłady rodzajów błędów:

- LekserError
  - błędy na poziomie tworzenia Tokenów.
- ParserError

    - błędy na poziomie tworzenia AST
    - nadpisywanie słów kluczowych
- SemanticError
    - wywoływanie funkcji z błędnymi typami argumentów
    - ciało funkcji bez wyrażenia `return`
    - redeklaracja zmiennej
- DevelopmentError - specjalny rodzaj błędu, którego końcowy użytkownik nigdy nie powinien zobaczyć. Wystąpienie tego rodzaju błędu oznacza popełnienie błędu przez programistę translatora np. gdy w definiowaniu węzła `Literal` przekażemy inny rodzaj tokenu niż token z wartością literalną. Kilkukrotnie uratował mnie od żmudnego szukania błędu w kodzie.

Wejście/wyjście

Przykład wykonania programu: `./translate_file file.cpp`

Skutkiem wykonania jest plik o ten samej nazwie, ale rozszerzeniu `.py`, dla przykładu powyżej: `file.py`

Dodatkowo zdefiniowany jest skrypt `compare_outputs.sh`, który służy do porównania wyników wykonania kodu napisane w C++ oraz przetłumaczonego do Pythona. Jego schemat działania:

1. skompiluj kod .cpp
2. wykonaj skompilowany kod oraz zapisz jego wynik to pliku `cpp_output.txt`
3. wykonaj przetłumaczony plik .py oraz zapisz jego wynik do pliku `py_output.txt`
4. porównaj pliki, możliwe odpowiedzi:
    - `OK - files are the same`
    - `ERROR - files are NOT the same`
5. usuń pliki: wykonywalny oraz wyniki programów

Testy

- Lekser
    - Proste testy jednostkowe sprawdzające czy lekser rozpoznaje wszystkie rodzaje tokenów oraz sekwencje tokenów.

      Przykład, czy ciąg znaków `int a = 10;` zostanie odpowiednio przetłumaczony na tokeny `type=int, id=a, assign_operator, literal=10`
- Parser
    - Proste testy jednostkowe sprawdzające czy lekser z parserem komunikują się w odpowiedni sposób. W powyższym przykładzie parser powinien zwrócić węzeł AST=`variableDeclaration` z podwęzłami `type=int`, `id=a` oraz `literal=10`
- Translator - testy całego systemu
    - Najbardziej złożone testy, ich przebieg jest następujący:
        1. Na wejściu zostaje podany kod programu w języku c++
        2. Zostaje zapisany plik źródłowy z podanym kodem
        3. Następuje analiza pliku wejściowego oraz translacja do kodu w języku Python
        4. Kod w języku c++ zostaje skompilowany

5. Skompilowany kod zostaje wykonany, a jego wynik zostaje zapisany do pliku tekstowego
6. Kod w języku Python także zostaje wykonany, a jego wynik zostaje zapisany do pliku tekstowego
7. Oba pliki tekstowe zostają porównane, jeżeli się różnią to test zostaje zakończony porażką
8. Wszystkie pliki wyprodukowane "po drodze" zostają usunięte

   Przykład - plik na wejściu:

   ```
   int main(){
       int i = 0;
       while(i<10){
           std::cout<<i<<std::endl;
           i = i+1;
       }
       return 0;
   }
   ```

   po skompilowaniu i wykonaniu skutkuje wypisaniem na standardowe wejście cyfr od 0 do 10 włącznie, każda w nowej linijce. Kod po translacji do języka Python powinien zachowywać się tak samo.

Przykład tłumaczenia

Kod wejściowy .cpp:

```
#include<string>
#include<iostream>

int fun(int arg1, int arg2){
    int i = 10;
    while( i > 0 ){
        std::cout<<i<<std::endl;
        std::cout<<"\n";
        i = i-1;
    }

    return 12;
}

int main(){
    fun(1,2);
    return 0 ;
}
```

Kod wynikowy .py:

```
def fun(arg1, arg2):
    i = 10
    while i > 0:
        print(i)
        print("\n", end="")
```

```
        i = i - 1
    return 12

def main():
    fun(1, 2)
    return 0


if __name__ == "__main__":
    main()
```