Kamil Boryczka
Metody translacji Temat nr 2: Proste C z typami int i double.

Funkcjonalność:

Deklarowanie zmiennych (dopuszczalne typy: int (64-bitowa liczba całkowita), double (liczba zmiennoprzecinkowa o rozmiarach double z języka C))
Przykład:
int a, b;
double c;

Konwersja typów
–int → double – może być niejawna
–double → int – dopuszczalna jest tylko jawna (przy pomocy odpowiedniego operatora).

Jest przyjęte założenie, że zadeklarowana zmienna żyje do końca bloku, w którym się znajduje. Jest także przyjęte założenie na potrzeby logiki (działającej na typie int), że 0 oznacza fałsz, a dowolna inna liczba prawdę.

Obsługa operatorów o różnym priorytecie (1 - najwyższy):

1. Operatory unarne prefiksowe (od prawej do lewej)
    - minus unarny
    ! negacja (dopuszczalna tylko dla typu int)
    (double) operator konwersji na typ double
    (int) operator konwersji na typ int
2. Operatory arytmetyczne mnożenia (od lewej do prawej)
    * mnożenie
    / dzielenie (jeżeli oba wyrażenia int, to całkowite (int), w.p.p. zmiennoprzecinkowe (double)
    % modulo
3. Operatory arytmetyczne dodawania (od lewej do prawej)
    + dodawanie
    - dzielenie
4. Operatory logiczne (od lewej do prawej)
    && koniunkcja
    || alternatywa
5. Operatory porównania (od lewej do prawej)
    == czy jest równe
    != czy nie jest równe
    > czy jest większe
    >= czy jest większe bądź równe
    < czy jest mniejsze
    <= czy jest mniejsze bądź równe
6. Operatory przypisania (od prawej do lewej)
    = przypisanie
    += zwiększenie wartości zmiennej o prawą stronę
    -= zmniejszenie wartości zmiennej o prawą stronę
    *= pomnożenie wartości zmiennej przez prawą stronę
    /= podzielenie wartości zmiennej przez prawą stronę
    %= wartość modulo prawa strona

Instrukcja blokowa {}
Nawiasowanie ()

Komentarze:
Są dopuszczalne dwa typy komentarzy, które są traktowane jako białe znaki:
// komentarz linii (kończy go znak końca linii)
/* */ komentarz wielolinijkowy
Białe znaki:
znak spacji, znak tabulatora, znak końca linii oraz komentarze. Są traktowane jako przerwy.

Instrukcje warunkowe:
warunek musi być typu int
Instrukcja warunkowa if (else i następujący po nim blok są opcjonalne). Dopuszczalny po nim jest blok bądź pojedyncza linia kodu (line w gramatyce).
Przykład:
```
int a=8;
if(a)
{
     a++;
}
else
{
     a--;
}
```

Pętla for – dopuszczalne są w nim, jak i w pętli while, break (służy do przerwania wykonania pętli) i continue (natychmiastowe przejście do końca bloku). Wartość początkowa parametru jest ustalana w 1. części, w 2. części sprawdzane przed wejściem do pętli, część 3 wykonywana na koniec pętli. Zmienna zadeklarowana w 1. części musi być wcześniej utworzona.
Przykład:
```
int i;
for(i=3;i<20;i++) {
     print(i);
     if(i==22) {break;}
}
```

Pętla while – wykonywana dopóty, dopóki warunek!=0
Przykład:
```
int a=5;
while (a<10) {
     a++;
     if(a==7) continue;
}
```

Funkcje:

Istnieje możliwość definiowania własnych funkcji. Funkcja może zwracać jeden z trzech typów: int, double lub void (brak zwracanego typu) oraz mogą przyjmować dowolną, acz z góry ustaloną liczbę parametrów typu int lub double. Parametry są przekazywane przez kopię. Funkcje muszą mieć unikatową nazwę oraz nie może istnieć funkcja o nazwach: print, scan, if, while, for, return.

Każda funkcja musi posiadać unikatową nazwę. Funkcje, poza funkcją o nazwie main, mogą być wywoływane rekursywnie.
Parametr jest zwracany poprzez return - natychmiastowe wyjście z funkcji zwracając wynik (dla funkcji „zwracającej” void brak wyniku, typ wyniku musi być zgodny z typem zwracanym przez funkcję bądź konwertowalny niejawnie)
Musi istnieć funkcja nieprzyjmująca żadnych parametrów i zwracająca int o nazwie main, która jest punktem startowym interpretera.
Przykład:

```
int function(int a, double b)
{
     return a;
}
```

Instrukcja wejścia:
scan() - wczytuje int z wejścia
scanf() - wczytuje double z wejścia

Instrukcja wyjścia:
print("{string}") - zwraca na wyjście stringa. Może być dodawany do int lub double tworząc string powstały z konkatenacji stringa oraz napisu odpowiadającemu danej liczbie int lub double.
Znaki specjalne stringa:

\n	znak końca linii
\t	znak tabulacji
\\	znak backslashu
\b, \r	zdefiniowane
\"	znak cudzysłowiu

Przykłady wykorzystania języka:

```
int square(int a) {
     return a*a;
}

int main() {
     int a=scan();
     for(int i=0;i<5;i++) {
          a+=square(i);
     }
     //program should print scan()+30
     print("Value: "+a);
}

//-------------------------------------
int Fibonacci(int a) {
     if(a==0 || a==1) {return 1;}
     return Fibonacci(a-1)+Fibonacci(a-2);
}

int main() {
     int a=scan();
```

```
     //if a>92, there will be error
     print("Value: "+Fibonacci(a));
}


//-------------------------------------
double pow(double a, int b) {
     int i=0;
     for(i=0;i<b;i++) {
          a*=b; //implicit conversion of b to double
     }
     return a;
}

int main() {
     print(pow(2.4,2)); //there should be 5,76 on output
}
```

Obsługa błędów:

–błędy na poziomie skanera – pokazuje błąd (błędny fragment tekstu) z numerem linii
–błędy na poziomie parsera – pokazuje typ błędy z numerem linii
–błędy na poziomie mapowania drzewa – pokazuje typ błędu
–błędy na poziomie interpretera – wyjątek typu std::runtime_error z odpowiednią wiadomością

Sposób testowania:

Testy jednostkowe za pomocą Google Test (dla każdego modułu z osobna). Jest przyjęte założenie przyrostowe, czyli testy danego modułu zakładają poprawność testów wcześniejszych etapów. Testy interpretera wykonują testy wykonania od podanego kodu źródłowego języka.

Sposób uruchomienia:

Program do interpretacji będzie pisany na Linuxie z użyciem kodu napisanego w języku C++20 kompilowanym z użyciem GCC.
./interpret file.cmm –> spróbuj skompilować plik kodu (rozszerzenie cmm) i uruchom
-s 128 -> maksymalna długość nazwy (domyślnie 128)
Wejście i wyjście: stdin i stdout w terminalu basha. Błędy będą wysyłane na stderr na terminalu.
-i -> jeżeli był błąd, zatrzymaj dalszą kompilację i wyjdź (domyślnie próbuje dalej)
Jeżeli nie zostanie podany parametr w uruchomieniu, wtedy kod będzie wczytany na podstawie stringa utworzonego z wejścia danych.

Zwięzły opis sposobu realizacji:

Moduły:
Moduł obsługi źródła danych – potrafi przekazać następny znak bądź informację o skończeniu się źródła. Posiada konkretyzacje dla niniejszych typów źródeł:
    Plik tekstowy (z użyciem fstream)
    Ciąg znaków (przekazywany za pomocą std::string)
Main - mikromoduł zajmujący się koordynacją działania modułów oraz wczytaniem flag uruchomienia programu

Moduł skanera (Scaner) - zajmuje się analizą leksykalną. przetwarza wejście (z Loadera, leniwie wczytywane) na tokeny (tablica obiektów klasy Token). Służy do rozbijania źródła danych na tokeny. Źródło danych będzie wczytywane znak po znaku. Po stwierdzeniu, że wczytane znaki tworzą prawidłowy token, jest on przekazywany do parsera, jeden po drugim, na prośbę od parsera. Jest wspierany przez moduł obsługi źródła danych.
Moduł parsera (Parser) – zajmuje się analizą składniową. Przetwarza tokeny otrzymane z modułu skanera na tablicę drzew składniowych funkcji oraz węzłów odpowiadających zmiennym globalnym.
Moduł mapowania (MappedSyntaxTree) – zajmuje się końcową fazą analizy składniowej. Przetwarza wcześniej uzyskaną tablicę funkcji i przekształca na słownik funkcji przesuwając węzły uzyskane w module parsera. Sprawdza poprawność nazw funkcji oraz to, czy istnieje odpowiednia funkcja main.
Interpreter – wykonuje operacje na drzewie uzyskanym z Parser. Korzysta z metod pomocniczych do wykonywania poszczególnych obiektów. Jako parametry przyjmuje wejście oraz wyjście, które musi być odpowiednio typu std::istream i std::ostream, gdzie dopuszczalne są ich klasy pochodne.

Struktury pomocnicze:

Tablica tokenów – tablica predefiniowana zawierająca wszystkie symbole, jakie powinien rozpoznawać skaner.
Moduł obsługi błędów – organizuje prezentację błędów na każdym etapie.
Tablica zdefiniowanych nazw – zajmuje się organizacją struktury zdefiniowanych przez użytkownika identyfikatorów.
Metoda wykonania dla każdego typu węzła w drzewie składniowym.

Typy tokenu:

string – ciąg znaków
literal – liczba int lub double
type_name – int lub double
id – nazwa literału (może zawierać nazwy funkcji scan lub scanf)
neg_op – operator negacji !
minus – operator – (może być unarny lub oznaczać
assign_op – operatory = += -= *= /= %=
logic_op – operatory || &&
rel_op – operatory porównania
add_op – znak +
mult_op – operatory * / %
conversion – operator konwersji (int) i (double)
if – instrukcja warunkowa if
else – druga część instrukcji warunkowej
while – pętla while
for – pętla for
type_name – nazwa typu zmiennej (int lub double)
error – niepoprawny token
return – instrukcja return
loop_mod – modyfikator pętli (break lub continue)
par_begin – znak (
par_end – znak)
end – znak ;
block_begin – znak {

block_end – znak }
comma – znak ,

Gramatyka języka:

```
//- zakres
//any - dowolny znak
//endl - znak końca linii
//* 0 lub więcej wystąpień
//+ 1 lub więcej wystąpień
//[] optional element

id = letter {digit | letter}* .
letter = "a"-"z" | "A"-"Z" | "_" .
digit = "0"-"9" .
comment = ("//" any* endl) | ("/*" any* "*/")
string =  "\"" any* "\"" .
int_number = {""} digit+ .
double_number = {"-"} digit* "." digit+ .
type_name = "int" | "double" .
literal = int_number | double_number .
fun_call = id arguments .
loop_mod = "continue" | "break"
par_begin = "(" .
par_end = ")" .
end = ";" .
if = "if" .
else = "else" .
while = "while" .
for = "for" .
Comma = "," .

neg_op = "!" .
minus = "-" .
assign_op = "=" | "+=" | "-=" | "*=" | "/=" | "%=" .
logic_op = "||" | "&&" .
rel_op = "==" | "!=" | "<" | ">" | "<=" | ">=" .
add_op = "+" .
mult_op = "*" | "/" | "%" .
conversion = "(double)" | "(int)"
block_begin = "{" .
block_end = "}" .


parenth = par_begin expression par_end .

expression = logic_expr {rel_op logic_expr} .
logic_expr = add_expr {logic_op add_expr } .
add_expr = mult_expr {(add_op | minus) mult_expr } .
mult_expr = un_expr {mult_op un_expr} .
un_expr = {minus | neg_op | conversion} prim_expr .
prim_expr = (literal | id | parenth | string | fun_call)
```

```
line = (init | assign | fun_call | loop_mod) end .
block = block_begin {if_st | while_st | for_st | return_st | line
| block } block_end .
cond_block = line | block .

program = {(init end) | function_def}
function_def = (type_name | "void") id par_begin parameters
par_end block .
parameters = type_name id {comma type_name id}*  .
arguments = par_begin [ expression {comma expression} ] par_end .

if_st = if par_begin expression par_end cond_block [else
cond_block] .
while_st = while par_begin expression par_end cond_block .
for_st = for par_begin [assign] end [expression] end [assign]
par_end cond_block .
init = typename id [assign_op expression ] .
assign = id assign_op expression .
return_st = return [expression] end
```