



AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

**WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI,
INFORMATYKI I INŻYNIERII BIOMEDYCZNEJ**

KATEDRA INFORMATYKI STOSOWANEJ

Teoria Kompilacji i Kompilatory

Implementacja interpretera do autorskiego języka dirCount

Autor:

Kierunek studiów:

Opiekun pracy:

Paweł Gałka, Wojciech Sałapatek, Maciej Nęcza, Marcin Grzyb

Informatyka

mgr inż. Paweł Jemioło

Kraków, 2020

Spis treści

1. Zespół i zakres prac	4
2. Wstęp Teoretyczny	5
2.1. Abstract Syntax Tree	5
2.2. Parser LL	6
2.3. Gramatyka języka	6
3. Typy	9
3.1. Int	9
3.2. Float	9
3.3. Char	10
3.4. String	10
3.5. Boolean	10
3.6. List	10
3.7. Dict	10
3.8. Function	10
4. Operacje	11
4.1. Operatory arytmetyczne	11
4.2. Operatory logiczne	12
4.3. Operatory dla zmiennych typu String	12
4.4. Operatory na Listach	12
4.5. Operatory na słownikach	12
5. Komendy	13
5.1. Declare	13
5.2. If	14
5.3. Let	14
5.4. While	14
5.5. For	14
5.6. Print	15

5.7. Input.....	15
5.8. Invoke function	15
6. Instrukcja.....	16
6.1. Przykładowe struktury programów	16
6.2. Instrukcja Uruchomiania	16
6.2.1. Kreator podstawowych zmiennych	16

1. Zespół i zakres prac

- Maciej Nędza - testy manualne, przykładowe programy
- Wojciech Sałapatek - architektura wstępna, testy integracyjne
- Marcin Grzyb - dokumentacja, utilities tworzenia kodu
- Paweł Gałka - development funkcjonalności

Wszystkie założenia projektu zostały spełnione. Interpreter został przetestowany automatycznie i manualnie.

Interpreter spełnia założenia obrane na początku projektu. Dodatkowo stworzone zostały utilsy do generowania kodu w tym języku.

2. Wstęp Teoretyczny

Dircount jest ezoterycznym językiem programowania. Język opiera się na hierarchicznej strukturze katalogów, aby naśladować strukturę parsowania. Podczas działania programu Foldery zmienne są przechowywane jako inny zestaw folderów (każdy z nazwą, typem i wartością), tym razem w folderze app-data użytkownika. Wszystkie Instrukcje w dircount są zdefiniowane przez liczbę katalogów w aktualnie rozpatrywanym folderze. Foldery są czytane w porządku alfabetycznym.

Celem tej pracy jest skonstruowanie interpretera analizującego język dirCount w języku programowania Python.

Interpreter jest wolny od elementu lexera, ponieważ sama struktura kodu jednoznacznie wyznacza tokeny które opisują dany kontekst, element.

2.1. Abstract Syntax Tree

Aby zrozumieć istotę działania tego języka trzeba zrozumieć koncepcję AST.

Abstrakcyjne drzewo składniowe (AST), lub po prostu drzewo składniowe, jest drzewną reprezentacją abstrakcyjnej struktury składniowej kodu źródłowego napisanego w języku programowania. Każdy węzeł drzewa oznacza konstrukcję występującą w kodzie źródłowym.

Składnia jest „abstrakcyjna” w tym sensie, że nie reprezentuje każdego szczegółu występującego w rzeczywistej składni, a jedynie szczegóły strukturalne lub związane z treścią. Na przykład nawiasy grupujące są niejawnie w strukturze drzewa, więc nie muszą być reprezentowane jako osobne węzły. Podobnie konstrukcję składniową, taką jak wyrażenie „warunek-to-wtedy”, można określić za pomocą pojedynczego węzła z trzema rozgałęzieniami.

Odnosząc tą definicję drzewa składniowego do języka dirCount, można powiedzieć że sama struktura "kodu" w języku dirCount jest zmodyfikowanym drzewem składniowym, gdzie jego budowa nie jest już abstrakcyjna tylko dokładna w sensie wartości zmiennych, itp. użytych w kodzie.

Zmianami w stosunku do teoretycznego AST jest też linkowanie węzłów z liśćmi głównie w momencie odnoszenia się referencjami do zmiennych. W języku dirCount aby odwołać się do zmiennej możemy albo podać jej nazwę, albo wprost podać miejsce zadeklarowania tej zmiennej odnosząc się do liścia AST, przez co struktura kodu jest dosyć złożona.

2.2. Parser LL

Parser LL to parser czytający tekst od lewej do prawej i produkujący lewostronne wyprowadzenie metodą zstępującą. Popularne rodzaje parserów LL to parsery sterowane tablicą i rekurencyjne.

Parsery klasy LL(k) parsują znak po znaku, utrzymując stos zawierający „spodziewane symbole”. Na początku znajdują się tam symbol startowy i znak końca pliku. Jeśli na szczycie stosu jest ten sam symbol terminalny, jaki aktualnie znajduje się na wejściu, usuwa się go ze szczytu stosu i przesuwa strumień wejściowy na kolejny znak. Jeśli inny symbol terminalny zwraca się błąd. Jeśli występuje tam jakiś symbol nieterminalny, to zdejmuje się go i zależnie od tego symbolu oraz od k kolejnych znaków wejścia, umieszcza na stosie odpowiedni zestaw symboli.

Parser LL(*) to lewostronny parser z podglądem dowolnej liczby symboli.

Odnosząc to do języka dirCount schemat interpretera z tematu projektu, można dość luźno powiązać z działaniem parsera LL(*) który w trakcie traversalu wykonuje działania związane z odczytem danych symboli reprezentowanych przez ilość folderów. Interpreter na danym poziomie parsingu podgląda od 1 do 2 podfolderów w zależności od kontekstu w którym się znajduje i wybiera odpowiednie działanie na podstawie zliczonej ilości podfolderów (np. w procesie declare opisanym w rozdziale 2, decyduje o typie deklarowanej zmiennej).

Na etapie parsingu każde wywołanie funkcji generuje utworzenie związanego z danym wywołaniem stosu zmiennych. W momencie każdej deklaracji, zmiany wartości dokonywanana jest aktualizacja zmiennej przechowywanej w strukturze słownikowej klucz : wartość = (path, name) = (value, type).

2.3. Gramatyka języka

Dla interpretera nie używaliśmy pregenerowanych parserów typu ANTLR, Yacc, Bison ze względu na problem formalizacji gramatyki na oczekiwane tokeny przez te kompilatory. Język dirCount w założeniu miał być wolny od kontekstu tekstowego i opierać swoje działanie jedynie na strukturze drzewiastej swojej reprezentacji kodu.

Poniżej przedstawimy bazowy zarys gramatyki naszego języka, który daje ogólny pogląd na język dirCount.

1. Root \rightarrow Command
2. Command \rightarrow CommandExpr Command | epsilon
3. CommandExpr \rightarrow CommandType CommandValue
4. CommandType \rightarrow CommandTypeValue

5. `CommandTypeValue` \rightarrow `->` `function-invoke` | `declare` | `let` | `if` | `while` | `for` | `print` | `input`
6. `CommandValue` \rightarrow `CommandDirFunction1` `CommandDirFunction2` | `CommandDirDeclare1` `CommandDirDeclare2` `CommandDirDeclare3` | `CommandDirLet1` `CommandDirLet2` | `CommandDirWhile1` `CommandDirWhile2` | `CommandDirIf1` `CommandDirIf2` `CommandDirIf3` | `CommandDirFor1` `CommandDirFor2` `CommandDirFor3` `CommandDirFor4` | `CommandDirPrint1`
7. `CommandDirFunction1` \rightarrow `FunctionsArgsList`
8. `CommandDirFunction2` \rightarrow `function-path-link`
9. `FunctionArgsList` \rightarrow `Element` `FunctionArgsList` | `epsilon`
10. `CommandDirDeclare1` \rightarrow `int` | `float` | `char` | `string` | `boolean` | `list` | `dict` | `function`
11. `CommandDirDeclare2` \rightarrow `CommandDirFunction1` `CommandDirFunction2` | `OperationDir` `OperationDirVal` `OperationDirVal` | `int0 .. int15` | `float0 .. float31` | `boolDir` | `char0 .. char7` | `string0` `string1` | `listDir .. listdir3` | `dictDir .. dictDir4` | `function0` `function1` `function2` `function3` `function4` `function5`
12. `function0` \rightarrow `Command`
13. `function1` \rightarrow `Element` `function1` | `epsilon`
14. `function2` \rightarrow `string0` `string1`
15. `string0` \rightarrow `char0 .. char7`
16. `listDir` \rightarrow `Element` `ListDir` | `epsilon`
17. `dictDir` \rightarrow `DirElement` `DictDir` | `epsilon`
18. `CommandDirDeclare3` \rightarrow `String0` `string1`
19. `CommandDirLet1` \rightarrow `LinkDir`
20. `CommandDirLet2` \rightarrow `CommandDirFunction1` `CommandDirFunction2` | `OperationDir` `OperationDirVal` `OperationDirVal` | `int0 .. int15` | `float0 .. float31` | `boolDir` | `char0 .. char7` | `string0` `string1` | `listDir .. listdir3` | `dictDir .. dictDir4`
21. `LinkDir` \rightarrow `link-path` | `link-path` `string0` `string1`
22. `OperationDir` \rightarrow `OperationSubtype` `OperationSubtypeValue`
23. `OperationSubtype` \rightarrow `arithmetic` | `compare` `compare` | `string` `string` `string` | `list` `list` `list` `list` | `dict` `dict` `dict` `dict`
24. `OperationSubtypeValue` \rightarrow operacje w sekcji Operacje

25. $\text{OperationValue} \rightarrow \text{CommandDirFunction1} \text{CommandDirFunction2} \mid \text{OperationDir} \text{OperationDirVal} \mid \text{int0} \dots \text{int15} \mid \text{float0} \dots \text{float31} \mid \text{boolDir} \mid \text{char0} \dots \text{char7} \mid \text{string0} \text{string1} \mid \text{listDir} \dots \text{listdir3} \mid \text{dictDir} \dots \text{dictDir4}$
26. $\text{CommandDirWhile1} \rightarrow \text{OperationDir} \text{OperationDirVal} \text{OperationDirVal}$
27. $\text{CommandDirWhile2} \rightarrow \text{Command}$
28. $\text{CommandDirIf1} \rightarrow \text{boolDir} \mid \text{OperationDir} \text{OperationDirVal} \text{OperationDirVal}$
29. $\text{CommandDirIf2} \rightarrow \text{Command}$
30. $\text{CommandDirIf3} \rightarrow \text{epsilon} \mid \text{Command}$
31. $\text{CommandDirFor1} \rightarrow \text{Command} \mid \text{CommandDirDeclare1} \text{CommandDirDeclare2} \text{CommandDirDeclare3}$
32. $\text{CommandDirFor2} \rightarrow \text{boolDir} \mid \text{OperationDir} \text{OperationDirVal} \text{OperationDirVal}$
33. $\text{CommandDirFor3} \rightarrow \text{Command} \mid \text{CommandDirLet1} \text{CommandDirLet2}$
34. $\text{CommandDirFor4} \rightarrow \text{Command}$
35. $\text{Element} \rightarrow \text{OperationDir} \text{OperationDirVal} \text{OperationDirVal} \mid \text{int0} \dots \text{int15} \mid \text{float0} \dots \text{float31} \mid \text{boolDir} \mid \text{char0} \dots \text{char7} \mid \text{string0} \text{string1} \mid \text{listDir} \dots \text{listdir3} \mid \text{dictDir} \dots \text{dictDir4}$
36. $\text{CommandDirPrint1} \rightarrow \text{Element} \text{CommandDirPrint1} \mid \text{epsilon}$

Foldery boolDir , intDirX , floatDirX , charDirX są postaci bitowej przyjmując wartość 1 gdy w środku istnieje folder.

Ramowa gramatyka języka dirCount nie jest lewostronnie rekursywna i nie potrzeba lewostronnej faktoryzacji dlatego translator opiera się o zmodyfikowane działanie parsera LL^*

3. Typy

Folder typu instrukcji deklaracji zmiennej zawiera informacje o typie zmiennej.

W zależności od N(liczba w nawiasie przy nazwie komendy) równemu liczbie podkatalogów folderu typu dostępne są następujące typy:

- int(1)
- float(2)
- char(3)
- string(4)
- boolean(5)
- list(6)
- dict(7)
- function(0)

Podkatalogi wartości zmiennej zawierają w sobie 1 lub 0 podfoleń - reprezentują one binarne 1 (w przypadku obecności katalogu) i 0 - w przypadku jego braku

3.1. Int

Folder wartości zmiennej typu int musi zawierać 16 podfoleń. Pierwszy podfoleń reprezentuje znak (0 dla liczby dodatniej, 1 dla ujemnej), pozostałe 15 folderów to zapis liczby w postaci binarnej.

3.2. Float

Zmienna typu float jest reprezentowana zgodnie ze standardem IEEE 754. Folder wartości zmiennej musi zawierać 32 podfoleń. Pierwszy podfoleń reprezentuje znak (0 dla liczby dodatniej, 1 dla ujemnej), 23 kolejnych folderów zawiera wartość mantysy, 8 kolejnych folderów zawiera wartość cechy.

3.3. Char

Folder wartości zmiennej typu float musi zawierać 8 podfolderów. Zawierają one reprezentację zmiennej w kodzie ASCII.

3.4. String

Folder wartości zmiennej typu String musi zawierać 2 podfoldery. Pierwszy z nich zawiera katalogi w których zakodowana jest wartość poszczególnych zmiennych typu char składających się na napis. Ich wartość jest zakodowana w 8 podfolderach analogicznie jak przy deklaracji zmiennej typu char. Drugi natomiast pozostaje pusty.

3.5. Boolean

Folder wartości zmiennej typu boolean musi zawierać 1 podfolder. Jeśli zawiera on jeden podfolder, to zmienna przyjmuje wartość true, w przeciwnym wypadku przyjmuje wartość false.

3.6. List

Folder wartości zmiennej typu list musi zawierać 4 podfoldery. Pierwszy z nich zawiera podfoldery przechowujące wartości kolejnych elementów listy.

3.7. Dict

Folder wartości zmiennej typu dict musi zawierać 5 podfolderów. Pierwszy z nich zawiera kolejne podfoldery będące elementami słownika. Każdy z nich musi zawierać 2 podfoldery - pierwszy z nich przechowuje wartość klucza słownika, natomiast drugi zawiera wartość elementu.

3.8. Function

Folder funkcji musi zawierać 6 podkatalogów. Pierwszy z nich zawiera listę komend wykonywanych przez funkcję, drugi z nich zawiera listę nazw argumentów przekazywanych do funkcji, trzeci z nich zawiera nazwę zwracanej zmiennej, w przypadku gdy funkcja ma sygnaturę void folder ten jest pusty, w przeciwnym wypadku po wykonaniu komand z pierwszego folderu jest zwracana zmienna z lokalnego stosu zmiennych funkcji.

4. Operacje

W języku dirCount zostały zaimplementowane również operatory:

- arytmetyczne(1)
- logiczne(2)
- operatory dla zmiennych typu String(3)
- operatory na Listach(4)
- operatory na Słownikach(5)

Folder operacji musi zawierać 2 podkatalogi:

- liczba katalogów w pierwszym podkatalogu definiuje typ operatora, którego chcemy użyć
- liczba katalogów w drugim podkatalogu określa konkretny operator z danego typu

4.1. Operatory arytmetyczne

Dostępnymi operatorami arytmetycznymi(które zostały określone poprzez wartość(N)) są:

- operator dodawania(1)
- operator odejmowania(2)
- operator mnożenia(3)
- operator dzielenia(4)
- operator potęgowania(5)
- operator dzielenia modulo(6)

4.2. Operatory logiczne

Dostępnymi operatorami logicznymi(które zostały określone poprzez wartość(N)) są:

- $x < y$ $x > y$ (1 oraz 2): logiczne operatory relacji zwracające true gdy x jest większy/mniejszy od y
- $x \leq y$ $x \geq y$ (3 i 4): logiczne operatory relacji zwracające true gdy x jest większy lub równy/mniejszy lub równy od y
- $x == y$ (5): logiczny operator relacji zwracający true gdy x i y są równe

4.3. Operatory dla zmiennych typu String

Dla zmiennych typu String dostępnymi operatorami(określonymi poprzez wartość(N)) są:

- operator konkatencji(1)
- operatory porównania($==$, $!=$)-(2 oraz 3)

4.4. Operatory na Listach

Operacjami które można zastosować na listach(określonymi poprzez wartość(N)) są:

- pobranie elementu z listy(1)
- łączenie dwóch list(2)
- dodanie elementu do listy(3)
- usunięcie elementu na danej pozycji(4)

4.5. Operatory na słownikach

Operacjami które można zastosować na słownikach(określonymi poprzez wartość(N)) są:

- pobranie elementu ze słownika
- aktualizacja wartości elementu
- usunięcie elementu ze słownika

5. Komendy

Folder komendy musi zawierać 2 podkatalogi:

- liczba katalogów w pierwszym podkatalogu definiuje komendę którą chcemy użyć
- liczba katalogów w drugim podkatalogu zawiera dalsze wyrażenie komendy

Dostępne komendy:

- `invoke-function(0)`
- `declare(1)`
- `if(2)`
- `let(3)`
- `while(4)`
- `for(5)`
- `print(6)`
- `input(7)`

5.1. Declare

Komenda `declare` pozwala na deklarację zmiennej. Drugi katalog komendy `declare` musi zawierać 3 podkatalogi:

- pierwszy zawiera informacje o typie zmiennej
- drugi zawiera informacje o wartości zmiennej
- trzeci zawiera informacje o nazwie zmiennej

5.2. If

If jest wyrażeniem warunkowym. Jego drugi katalog musi zawierać 2 lub 3 foldery:

- pierwszy zawiera logiczny warunek
- drugi zawiera kod wykonywany w przypadku spełnienia warunku
- trzeci (opcjonalny) zawiera kod wykonywany w przypadku nie spełnienia logicznego warunku

5.3. Let

Let pozwala na zaktualizowanie wartości zmiennej podanej w linku symbolicznym. Jego drugi katalog musi zawierać 2 foldery:

- pierwszy przetrzymuje folder linkujący zmienną na 2 sposoby
 - link do zmiennej
 - link, foldery stringa z nazwą zmiennej
- drugi zawiera nową wartość zmiennej

5.4. While

While wykonuje instrukcje tak długo jak spełniony jest logiczny warunek. Jego drugi katalog musi zawierać 2 foldery:

- pierwszy zawiera logiczny warunek
- drugi zawiera kod wykonywany tak długo jak spełniony jest warunek

5.5. For

W zależności od liczby folderów w drugim katalogu komendy zdefiniowane są różne warianty pętli.

- W przypadku 1 podfolderu - Nieskończona pętla for, wykonywany jest kod z 1 podfolderu
- W przypadku 3 podfolderów - Pętla z warunkiem i definiowaną zmienną iteracyjną:
 - pierwszy podfolder zawiera deklarację zmiennej
 - drugi podfolder zawiera warunek logiczny
 - trzeci podfolder zawiera kod wykonany w przypadku spełnienia warunku

- W przypadku 4 podfolderów - Pełna pętla for, zawierająca komendę przypisania wykonywaną pod koniec każdej iteracji
 - pierwszy podfolder zawiera deklarację zmiennej
 - drugi podfolder zawiera warunek logiczny
 - trzeci podfolder zawiera instrukcję let - przypisania wartości do zmiennej
 - czwarty podfolder zawiera kod wykonany w przypadku spełnienia warunku

5.6. Print

Służy wypisaniu tekstu na ekran. Drugi katalog komendy musi zawierać jeden podfolder przetrzymujący listę argumentów, które zostaną wypisane na ekran.

5.7. Input

Służy pobraniu wartości zmiennej od użytkownika. Przy wykonaniu tworzony jest katalog input w katalogu głównym programu, do którego użytkownik powinien przemieścić folder zawierający deklarację zmiennej - identyczny z drugim katalogiem komendy declare. Po stworzeniu zmiennej katalog input zostaje usunięty, a wartość zmiennej zostaje zapisana w słowniku. Można się do niej odwołać poprzez link do drugiego katalogu komendy input.

5.8. Invoke function

Wywołanie funkcji musi zawierać 2 podkatalogi. Pierwszy z nich zawiera listę argumentów z jaką wywołujemy funkcję, drugi z nich jest linkiem do deklaracji funkcji odwołującym się do stosu zewnętrznej funkcji lub globalnej w przypadku niepowodzenia wyszukiwania w stosie funkcji matki.

6. Instrukcja

6.1. Przykładowe struktury programów

6.2. Instrukcja Uruchomiania

Aby uruchomić przykładowy program należy uruchomić skrypt *main.py*. Przykładowa komenda uruchamiająca interpreter wygląda:

```
python main.py ścieżka_do_głównego_katalogu_programu
```

Udostępniono również tryb debugowania, który pozwala na dokładną obserwację wydarzeń, które dzieją się podczas interpretacji programu. Aby uruchomić interpretację w trybie debugowania po nazwie skryptu(*main.py*) należy dopisać *-d*. Dodatkowo istnieje możliwość zapisu logów do pliku, komenda uruchamiająca program wygląda wtedy:

```
python main.py -df ścieżka_docelowa_pliku_log ścieżka_do_głównego_katalogu_programu
```

6.2.1. Kreator podstawowych zmiennych

Aby przyspieszyć żmudne tworzenie programów stworzono skrypt automatyzujący tworzenie zmiennych typów:

- int
- float
- boolean
- char
- string

Są one tworzone w formacie gotowym do deklaracji tj. katalog zawierający 3 podkatalogi które określają typ, wartość oraz nazwę zmiennej. Funkcjonalność udostępniona jest w skrypcie */utilities/variable_creator.py*. Korzystanie ze skryptu jest bardzo proste przykładowa komenda:

```
python variable_creator.py sciezka_w_ktorej_powstanie_zmienna int 25 a
```

jest odpowiednikiem `int a=25`.