

Dokumentacja końcowa z przedmiotu Techniki Kompilacji

Radosław Radziukiewicz

Wstęp

W ramach projektu zaprojektowano, zaimplementowano i przetestowano interpreter języka wspierającego typ macierzowy. Interpreter składa się z 3 modułów: analizatora leksykalnego, składniowego oraz właściwego interpretera który jednocześnie pełni rolę analizatora semantycznego.

Dodatkowo, w ramach projektu dostarczony został specjalny model obsługi wyjątków oraz biblioteka standardowa udostępniająca przydatne funkcje wejścia/wyjścia oraz operacje na macierzach.

Paradygmaty języka i typowanie

Stworzony język wspiera paradygmat proceduralny imperatywny. Typowanie język jest dynamiczne. Zmienne w języku są mutowalne a ich typ zależy od wykonania programu. Zakres widoczności zmiennych jest statyczny (leksykalny). W języku nie występuje mechanizm przestaniania.

Gramatyka

Poniżej zaprezentowano ostateczną gramatykę stworzonego języka.

```
program          = { funcDefinition }

funcDefinition   = identifier "(" parameters ")" statementBlock

parameters       = [ identifier { "," identifier } ]

statementBlock   = "{" {
                    ifStatement |
                    untilStatement |
                    returnStatement |
                    assignOrFuncCall |
                    statementBlock
                  } "}"

returnStatement  = "return" [ addExpression ]

ifStatement      = "if" "(" orCondition ")" statementBlock
                  [ "else" ( ifStatement | statementBlock ) ]

untilStatement   = "until" "(" orCondition ")" statementBlock

assignOrFuncCall = identifier (
                              ( "(" arguments ")" ) |
```

```

                                ( [ indexOperator ] "=" addExpression )
                                )

arguments                      = [ addExpression { "," addExpression } ]

addExpression                  = mulExpression { ("+" | "-") mulExpression }
mulExpression                  = atomicExpression { ("*" | "/" )
                                atomicExpression }
atomicExpression               = ["-"] ( identOrFuncCall |
                                literal |
                                "(" orCondition ")"
                                )

identOrFuncCall                = identifier [ "(" arguments ")" |indexOperator]
indexOperator                  = "[" selector "," selector "]"
selector                       = ( ":" | addExpression )

orCondition                    = andCondition { "or" andCondition }
andCondition                   = relCondition { "and" relCondition }
relCondition                   = ["!"] addExpression [ relOperator
                                addExpression ]

literal                        = matrixLiteral | numberLiteral | string
matrixLiteral                  = "[" addExpression { ( "," | ";" )
                                addExpression } "]"

numberLiteral = (0|[1-9][0-9]*) (.[0-9]*)?
identifier    = [a-zA-z][a-zA-Z0-9_-]*
string        = "[Alfa-numeric characters]*"

```

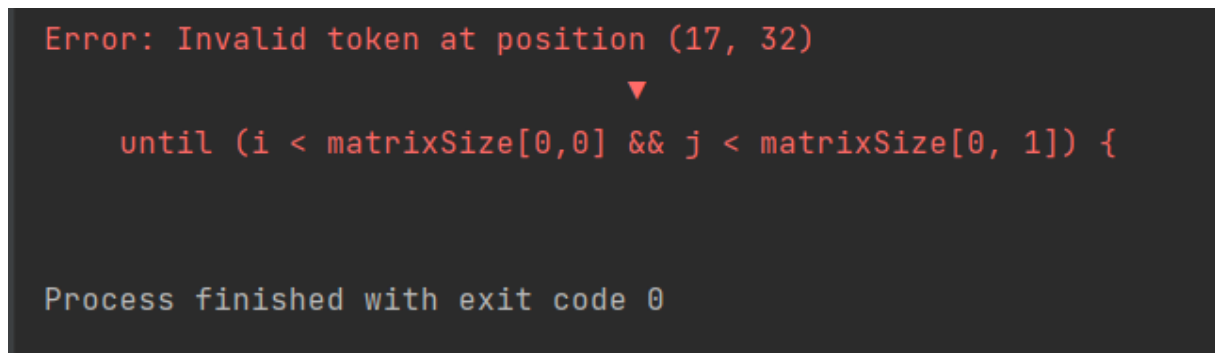
Analizator leksykalny

Pierwszym modułem powstałym w ramach implementacji interpretera jest analizator leksykalny. Analizator ten rozpoznaje następujące żetony, wykorzystywane do dalszej analizy kodu źródłowego programu.

- Identyfikator
- If, else, until, return
- Numery oraz napisy
- Operatory logiczne
- Wszystkie operatory porównania
- Operacje arytmetyczne
- Wszystkie rodzaje nawiasowania
- Przecinek, przypisanie, dwukropek oraz średnik
- Koniec tekstu

W przypadku żetonów napisu oraz liczby, analizator dokonuje konwersji ciągu znaków otrzymanych ze źródła na odpowiednie wartości napisowe i liczbowe. Wykrywane są również potencjalne sytuacje wyjątkowe, takie jak deklaracja zbyt długiego napisu czy zbyt wielkiej liczby.

Analizator leksykalny jest w pełni odporny na występowanie sytuacji wyjątkowych w pozyskanym kodzie źródłowym programu. Posiada on bogatą klasę wyjątków określającą przyczynę błędu wraz z informacjami diagnostycznymi ułatwiającymi prostą detekcję źródła anomalii przez użytkownika.



```
Error: Invalid token at position (17, 32)
      ▼
      until (i < matrixSize[0,0] && j < matrixSize[0, 1]) {

Process finished with exit code 0
```

Rysunek 1. Przykładowa informacja diagnostyczna

W ramach testowania poprawności implementacji analizatora leksykalnego przygotowano 16 testów. Większość testów zawiera kolejne podprzypadki testowe, zatem łączna liczba testów przekracza 50. Testy składają się z zarówno przypadków pozytywnych, jak i negatywnych.

Analizator składniowy

Drugim modułem powstałym w ramach implementacji interpretera jest analizator składniowy. Celem tego modułu jest wytworzenia drzewa dokumentu programu (tzn. wytworzenie drzewa programu ale bez zbędnych „pustych” konstrukcji). Drzewo dokumentu może zawierać następujące konstrukcje:

- Cały program
- Definicja funkcji
- Blok instrukcji
- Konstrukcja warunkowa
- Konstrukcja pętli
- Konstrukcja powrotu
- Wywołanie funkcji
- Przypisanie
- Operacje dodawania, mnożenia i zanegowane wyrażenia atomowe

- Warunki koniunkcji, alternatywy i relacji
- Stała macierzowa, napisowa i liczbowa
- Identyfikator
- Operator indeksowania i wybór kropkowy

Analizator składniowy operuje na ciągu żetonów dostarczonych przez analizator leksykalny i na ich podstawie tworzy odpowiednie konstrukcje składniowe.

Analizator składniowy jest w pełni odporny na wystąpienie sytuacji wyjątkowych. Każda potencjalna anomalia zgłaszana jest poprzez wykorzystanie odpowiedniej klasy wyjątków zawierającej opis zdarzenia, wraz z informacjami pozwalającymi na lepszą diagnostykę przyczyny zjawiska.

```
Error: Expected selector but got Token: type=COMMA, value=,, position=(33, 10) in context SyntacticContext.IndexOperator
▼
b[0, ,] = [10, 12, 13]

Process finished with exit code 0
```

Rysunek 2. Przykładowa informacja diagnostyczna

W ramach testowania poprawności dostarczonego analizatora składniowego stworzono 34 testy. Warto zaznaczyć tutaj, iż większość testów składa się z kilku przypadków podtestowych, dlatego faktyczna, łączna liczba testów przekracza 90. Testy składają się z zarówno przypadków pozytywnych, jak i negatywnych.

Interpreter

Ostatnim modułem bezpośrednio implementującym funkcjonalności związane z interpretacją języka jest moduł interpretera. Interpreter stosuje wzorzec odwiedzającego w celach ewaluacji programu. Wzorzec ten wymusza implementację dodatkowych metod w ramach drzewa dokumentu wytworzonego przez analizator składniowy. Metody te to metody akceptacji.

Interpreter pełni również rolę analizatora semantycznego. Wykrywa błędy związane z nieodpowiednią konwersją typów lub operacji arytmetycznych. Podobnie jak w przypadku poprzednich modułów jest w pełni odporny na wystąpienie sytuacji wyjątkowych.

```
Error: Invalid type VariableType.STRING
Stack trace:
evaluate program
evaluate function main
evaluate statement block
evaluate until statement
evaluate and condition
evaluate rel condition

Process finished with exit code 0
```

Rysunek 3. Przykładowa informacja diagnostyczna

W ramach testowania poprawności implementacji interpretera dostarczono 38 testów. Wiele testów składa się z kilku podtestów, dlatego faktyczna liczba testów przekracza 80. Testy składają się z zarówno przypadków pozytywnych, jak i negatywnych.

Moduły dodatkowe

W ramach implementacji rozwiązania dostarczono również kilka modułów pomocniczych. W tej sekcji znajduje się krótki opis każdego z nich.

Moduł obsługi źródła danych

Moduł ten obsługuje pobieranie kodu źródłowego z różnych abstrakcji wejścia. Dostarczono implementacje obsługi pliku oraz napisu. W ramach modułu istnieją 3 interfejsy:

- Surowe źródło danych. Jest ono odpowiedzialne za bezpośredni dostęp do pliku lub napisu i pobieranie następnego znaku w niezmienionej formie.
- Źródło unifikujące. Jego zadaniem jest konwersja znaków do jednolitej postaci. W szczególności, konwersji podlega znak końca linii będący różnych dla różnych typów systemów operacyjnych (np. Windows vs UNIX).
- Źródło pozycyjne. Źródło to dodatkowo „pamięta” aktualną pozycję w pliku. Jest ono przydatne do pozyskiwania informacyjnych komunikatów diagnostycznych przez analizator leksykalny.

Moduł obsługi wyjątków

Moduł odpowiedzialny jest za obsługę sytuacji wyjątkowych powstałych w trakcie interpretacji programu. Obsługuje on wszystkie możliwe wyjątki generowane przez dowolny moduł interpretera. Ponadto odpowiada on za czytelne i informacyjne przedstawienie przyczyny zajścia danego zdarzenia.

Moduł obsługi stosów

Moduł ten jest wykorzystywany przez interpreter w celach zarządzania widocznością i typem zmiennych. Zawiera implementację logiki pozyskiwania i ustawiania wartości zmiennych w ramach wykonania funkcji oraz otwierania kontekstów w ramach wywołania nowej funkcji.

Biblioteka standardowa

Do programu dołączona została mała biblioteka standardowa. Zawiera ona następujące funkcje dostępne do użytku:

- „print” – dokonuje wydruku parametrów na standardowe wyjście
- „cin” – umożliwia odczyt liczby
- „size” – informuje o wymiarach macierzy
- „transpose” – dokonuje transpozycji macierzy
- „ident” – tworzy macierz identyczności
- „full” – tworzy macierz o podanym kształcie wypełnioną daną liczbą
- „reshape” – zmienia kształt macierzy na nowy

Technologie

Implementacja interpretera wykorzystuje język programowania Python. Testy jednostkowe zostały wykonane przy wykorzystaniu biblioteki unittest. Operacje na macierzach zostały zaimplementowane przy wykorzystaniu biblioteki numpy. Dodatkowo, w ramach projektu stworzony został mechanizm ciągłej integracji poprzez wykorzystanie narzędzia GitHub Actions.