Techniki kompilacji - dokumentacja końcowa

Prowadzący - dr inż. Piotr Gawkowski

Tomasz Nowak

1 Opis języka

1.1 Wstęp

"LOGO z L-systemem" - interpreter języka *LTurtle* pozwalającego definiować L-system i opisywać elementarne operacje w tym systemie w języku "grafiki żółwia".
Podstawowe elementy języka:

- Funkcje wbudowane, zmieniające stan rysującego żółwia
- Zbiór symboli, każdy z nich reprezentuje pewien ciąg operacji żółwia
- Zbiór produkcji, generujących ciągi symboli w L-systemie

Dla uproszczenia, jedynym rodzajem zmiennych występujących w języku są symbole L-systemu (nie ma np. zmiennych liczbowych czy napisów). Są one jednocześnie jedynym rodzajem funkcji tworzonych przez użytkownika LTurtle. Ten język można nazwać językiem funkcyjnym, z wyjątkiem wbudowanych operacji wykonywanych przez żółwia. Składnia języka jest połączeniem konwencji z C (klamry, średniki) z Pythonem (przypisanie wartości do zmiennej jest równoznaczne z jej definicją, nie ma deklaracji typu; są jednolinijkowe komentarze rozpoczynające się #).

1.2 Operacje żółwia

Niektóre z poniższych funkcji przyjmą argumenty liczbowe. Liczby muszą być podane jawnie (nie ma zmiennych liczbowych). Możliwe "typy danych": int, float. Operacje dostępne są wewnątrz definicji symbolu L-systemu, a także poza nią.

- *forward(float+)* - przesuwa żółwia, rysując linię
- *rotate(float[-180.0:180.0])* - obrót (kąt dodatni - zgodnie z ruchem wskazówek zegara, ujemny - przeciwnie)
- *penup()* - wyłącza pisak (ruch żółwia nie powoduje rysowania)
- *pendown()* - odwrotność penup()

- *pencolour(int[0:255],* int[0:255], int[0:255]) - ustawia kolor (RGB)
- *goto(float, float)* - przesuwa żółwia na współrzędne określone bezwzględnie
- *pensize(float+)* - ustawia średnicę pisaka
- *scale(float+)* - wszystkie wielkości rysowane przez żółwia będą skalowane
- *pushstate()* - zapisuje na stosie obecny stan żółwia
- *popstate()* - zdejmuje ze stosu obecny stan żółwia

Operacja *scale* jest potrzebna, ponieważ funkcje definiowane w L-systemie nie przyjmują żadnych argumentów - wszystkie długości są zapisane jawnymi stałymi liczbowymi. Dzięki tej operacji można wielokrotnie wykorzystywać raz zdefiniowaną funkcję do rysowania obiektów różnych rozmiarów (przykład - test_code/fractal_plant - po zdefiniowaniu fraktala *plant* i narysowaniu zielonej rośliny, tworozna jest mniejsza czerwona roślina - z wykorzystaniem tej samej zmiennej *plant*). Do zmiany położenia na obrazie wynikowym i obrotu można wykorzystać operacje *rotate* i *forward*. Jedynym kształtem elementarnym używanym do budowy obrazków jest linia prosta (a tak naprawdę prostokąt - ze względu na grubość), ale możliwe byłoby dodanie innych operacji do rysowania bardziej skomplikowanych kształtów, np. *draw_circle* (po dodaniu odpowiedniego słowa kluczowego do gramatyki i zaimplementowaniu operacji rysowania).

1.3 Operacje na symbolach L-systemu

- Definicja nowego symbolu, np.: line = { forward(1.0); rotate(60.0); };
- Zdefiniowanie reguły podstawienia, np. X −> X + line;
- Ewaluacja wyrażenia określoną liczbę iteracji (z wykorzystaniem istniejących podstawień), podstawiane pod nowy symbol, np.: operation = evaluate(5, X+X+X);
- Wykonanie operacji przez żółwia, np.: execute(operation);

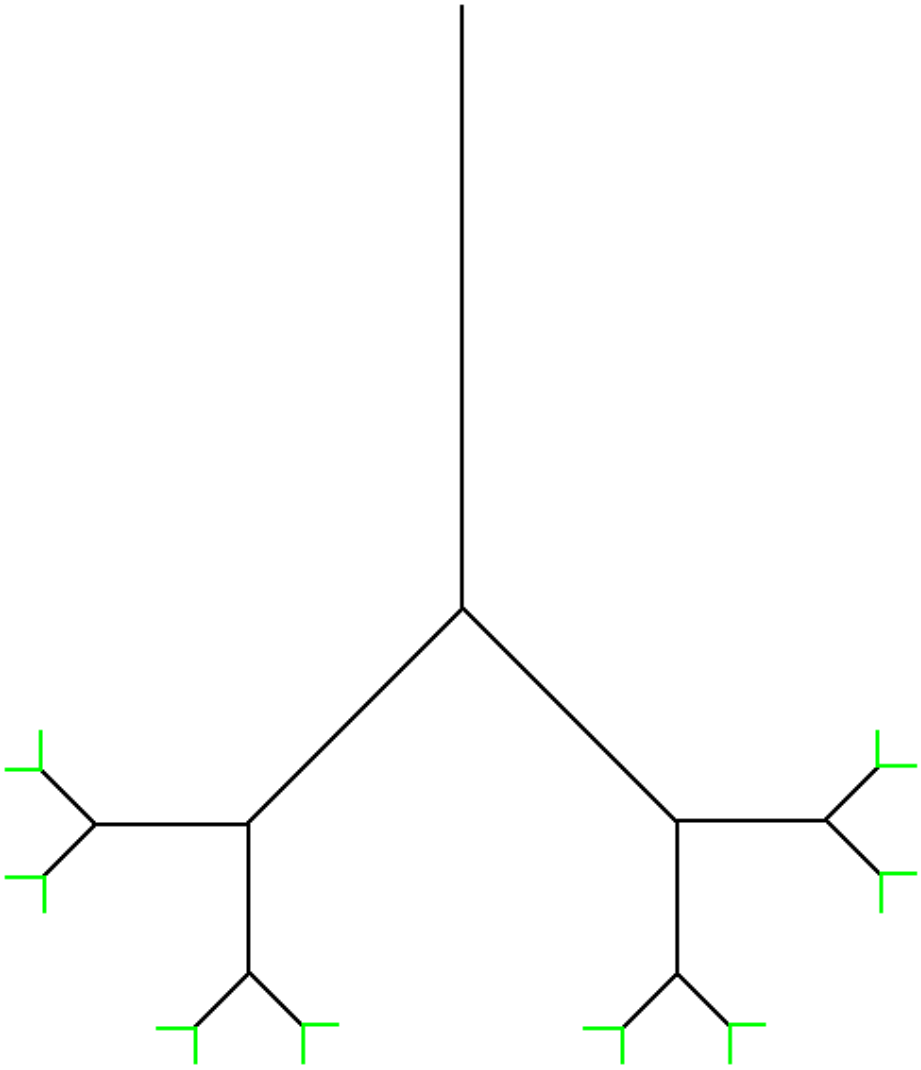
Operacje przypisania znaczenia danego symbolu (=) i zdefiniowania podstawienia (−>) mogą nadpisywać poprzednią wartość tylko gdy zostaną poprzedzone słowem kluczowym *redefine*, np.: *redefine line = { forward(1.0); rotate(30.0); };*. Operacja "=" musi być zdefiniowana dla każdego używanego symbolu, natomiast "−>"nie musi być zdefiniowana - symbol bez zdefiniowanych podstawień jest uznawany za terminalny.

1.4 Przykład kodu

Przykład z angielskiej Wikipedii o L-systemach - *Fractal (binary) tree* (zamienione nazwy symboli: s0 zamiast 0, s1 zamiast 1, left zamiat [, right zamiast]):

```
# based on an example on Wikipedia
s0 = { pencolour(0, 255, 0); forward(20.0); }; # 0 (leaf)
s1 = { pencolour(0, 0, 0); forward(40.0); }; # 1 (non-leaf)
left = { pushstate(); rotate(-45.0); }; # [ from Wikipedia
right = { popstate(); rotate(45.0); }; # ] from Wikipedia
s1 -> s1 + s1;
s0 -> s1 + left + s0 + right + s0;
tree4 = evaluate(4, s0); # use productions given above 4 times
pensize(2); # change lines width
execute(tree4); # draw using instructions stored in tree4
```

Wynikowy obrazek:



Inne przykłady są dostępne w katalogu test_code, a obrazy wynikowe - w katalogu images.

2 Formalna specyfikacja

2.1 Zdefiniowane tokeny

- *error* - pomocniczy, w poprawnym kodzie nieużywany
- *end_of_text*
- *int_number* - liczba całkowita to: cyfra niezerowa i ciąg cyfr, dozwolony też '-' na początku
- *float_number* - na początku liczba całkowtia, potem '.' i ciąg cyfr
- *literal* - litera, ciąg złożony z liter / cyfr / podkreślników
- (...)_*keyword*: forward, rotate, penup, pendown, pencolour, goto, pensize, scale, pushstate, popstate, evaluate, execute, redefine
- *production_operator* $->$
- *equals_symbol* =
- *plus_symbol* +
- *l_curly_bracket_symbol* {
- *r_curly_bracket_symbol* }
- *l_round_bracket_symbol* (
- *r_round_bracket_symbol*)
- *semicolon_symbol* ;
- *colon_symbol* ,

2.2 Gramatyka

Gramatyka opisana po usunięciu komentarzy (jednolinijkowy komentarz zaczynający się od #). Poniżej - opis gramatyki przed faktoryzacją (bardziej intuicyjny).

```
1: Program = { Statement semicolon_symbol } end_of_text
2: Statement = Definition | Execution
3: Definition = Redefinition | CreateDefinition
4: Execution = LiteralExecution | TurtleStatement
5: LiteralExecution = execute_keyword l_round_bracket_symbol
                      LiteralString r_round_bracket_symbol
```

```
6: Redefinition = redefine_keyword CreateDefinition
7: CreateDefinition = Operation | Production | Evaluation
8: LiteralString = { literal plus_symbol } literal
9: Operation = literal equals_symbol l_curly_bracket_symobl
               { TurtleStatement semicolon_symbol } r_curly_bracket_symbol
10: Production = literal production_operator LiteralString
11: Evaluation = literal equals_symbol evaluate_keyword
                 l_round_bracket_symbol int_number colon_symbol
                 LiteralsString l_round_bracket_symbol
12: TurtleOperation = TurtleOperationKeyword l_round_bracket_symbol
                      TurtleOperationArguments r_round_bracket_symbol
13: TurtleOperationKeyword = forward_keyword | rotate_keyword |
                             penup_keyword | pendown_keyword |
                             pencolour_keyword | goto_keyword |
                             pensize_keyword | scale_keyword |
                             pushstate_keyword | popstate_keyword
14: TurtleOperationArguments = { Number colon_symbol }
15: Number = int_number | float_number
```

Różnica w stosunku do dokumentacji wstępnej - zmiana konwencji nazewnictwa i przeniesienie analizy argumentów TurtleOperation na poziom semantyki (parser zaakceptuje dowolny ciąg liczb jako argumenty, a ich poprawność jest sprawdzana poza parserem).

Gramatyka po faktoryzacji - łatwa do parsowania (gramatyka regularna):

```
1: Program = { Statement semicolon_symbol } end_of_text
2: Statement = redefine_keyword Redefinition | literal Definition |
               execute_keyword LiteralExecution |
               TurtleOperationExecution
3: Redefiniton = literal Definition
4: Definition = production_operator Production |
                equals_symbol OperationOrEvaluation
5: LiteralExecution = l_round_bracket_symbol LiteralString
                      r_round_bracket_symbol
6: TurtleOperationExecution = TurtleOperation
7: Production = LiteralsString
8: OperationOrEvaluation = l_curly_bracket_symbol Operation |
                           evaluate_keyword Evaluation
9: Operation = { TurtleOperation semicolon_symbol } r_curly_bracket_symbol
10: Evaluaton = l_round_bracket_symbol int_number
                colon_symbol LiteralString l_round_bracket_symbol
11: LiteralString = { literal plus_symbol } literal
12: TurtleOperation = TurtleOperationKeyword l_round_bracket_symbol
                      TurtleOperationArguments r_round_bracket_symbol
13: TurtleOperationKeyword = forward_keyword | rotate_keyword |
                             penup_keyword | pendown_keyword |
```

```
                                    pencolour_keyword | goto_keyword |
                                    pensize_keyword | scale_keyword |
                                    pushstate_keyword | popstate_keyword
14: TurtleOperationArguments = { Number colon_symbol }
15: Number = int_number | float_number
```

3 Sposób uruchamiania

Wymagania: CMake, kompilator C++ (standard co najmniej 11), biblioteka BOOST (do testów jednostkowych), biblioteka SFML (do tworzenia grafiki). Przykład kompilacji w systemie Linux:

```
rm -rf build
mkdir build
cd build
cmake .. -DCMAKE_BUILD_TYPE=Release
make -j$(nproc)
```

Sposób uruchamiania:
[ścieżka do programu lturtle] [nazwa pliku wyjściowego]
Dane wejściowe (kod do interpretacji) przyjmowane są ze standardowego wejścia. Przykład uruchomienia programu, z przekierowaniem pliku wejściowego:

```
./build/lturtle images/bintree.png <test_code/binary_tree
```

4 Opis sposobu realizacji

4.1 Wykorzystane narzędzia

Program został napisany i przetestowany z wykorzystaniem poniższych narzędzi, ale powinien być przenośny.

- cmake 3.11.1
- gcc 7.3.1
- boost 1.66.0-2
- SFML 2.5.0-1
- system operacyjny: Manjaro Linux, procesor x86-64

4.2 Moduły programu

- exception - zawiera własne klasy wyjątków z "wygodnymi" w użyciu konstruktorami

- lexer - zawiera pliki source (moduł obsługi źródła), token (struktury danych opisujące pojedynczy token), lexer (analiza leksykalna kodu)
- parser - zawiera pliki syntax_tree (opis AST poniżej) i parser (zwraca odpowiednio powiązane ze sobą struktury danych z syntax_tree)
- semantics - moduł analizy semantycznej, opisany poniżej
- main - zawiera funkcję, od której zaczyna się wykonanie głównego programu (jest to oddzielone od reszty programu, aby funkcja main nie była linkowana do testów)
- tests - testy jednostkowe modułów: lexer, parser, semantics korzystające z biblioteki boost

4.3 Opis działania

Działanie analizatora leksykalnego: lekser odczytuje ze źródła kolejne znaki (przy czym po przetworzeniu danego znaku pobiera już kolejny) i zwraca tokeny wymienione w rozdziale 2.1. Token posiada informacje istotne dla semantyki programu (rodzaj tokenu i, w przypadku tokenów literal, int_number, float_number - wartość) oraz informację o tym, gdzie się zaczyna i kończy (co jest używane przy podawaniu komunikatu błędu). Uwaga - token int_number może być traktowany też jako float_number - wbudowane "rzutowanie". Słowa kluczowe są przechowywane w tablicy haszującej, przy czym wykorzystywana jest "doskonała" funkcja haszująca (brak kolizji i nadmiarowych miejsc w tablicy haszującej - parametry tej funkcji znaleziono w pomocniczym programie hashtest).

Działanie parsera - ogólna zasada działania jest bardzo podobna do analizatora leksykalnego (ze względu na prostotę języka - gramatyka regularna), z tą różnicą, że z gotowych tokenów budujemy struktury AST.

Struktury danych używane w AST:

- Program - ciąg złożony z obiektów dziedziczących po Statement
- Statement - typ abstrakcyjny
- Definition : public Statement - typ abstrakcyjny
- Operation : public Definition - tworzona np. w kodzie: x = forward(1);;
- Production : public Definition - tworzona np. w kodzie: x − > x + a;
- Evaluation : public Definition - tworzona np, w kodzie: a = evaluate(2, x + x);
- LiteralExecution : public Statement - tworzona np. w kodzie: execute(x);

- TurtleOperationExecution : public Statement - tworzona np. w kodzie popstate(); (operacja żółwia poza definicją zmiennej)
- LiteralString, np. x + a
- TurtleOperation - jedna z operacji zółwia
- TurtleOperationArguments - argumenty wewnątrz operacji żółwia, np. (255, 0, 0) wewnątrz pencolour()

Działanie analizatora semantycznego - jest podzielone na 2 części. Pierwsza, następująca równolegle z parsowaniem (jest to możliwe, ponieważ parsowanie kolejnych 'Statement' jest niezależne - unikanie zapisywania AST całego programu to oszczędność pamięci), zapamiętuje kolejne instrukcje żółwia i współrzędne, na których się znalazł, ale niczego nie rysuje. Dzięki temu rozdzielczość obrazu będzie dostosowywana dynamicznie do zakresu współrzędnych, po których przemieszczał się żółw. Rozdzielczość musi być obliczona przed rasteryzacją, ponieważ przyjęcie odgórnie narzuconej rozdzielczości wpływałoby negatywnie na jakość tworzonych obrazów albo znacząco ograniczałoby kreatywność użytkownika w tworzeniu obrazu. W drugiej fazie wykonania (w tym momencie plik z kodem programu może już być zamknięty) następuje rasteryzacja (jedyne, co jest wtedy potrzebne w pamięci, to sam obraz, i ciąg operacji żółwia - każdy program zapisany w LTurtle można sprowadzić do liniowej sekwencji operacji żółwia).

Klasy używane podczas analizy semantycznej:

- Interpreter - tworzy pozostałe klasy i wywołuje ich główne metody.
- CodeAnalyzer - klasa używana w pierwszej części analizy semantycznej. Wywołuje parser i interpretuje kolejne 'Statement'. Skutkiem interpretacji Statement może być:
  - stworzenie lub zmiana definicji (redefine) zmiennej opisanej przez ciąg operacji żółwia ('Operation') lub ewaluację innej zmiennej ('Evaluation') (implementacja tego pierwszego jest trywialna, natomiast stworzenie ewaluacji już nie, ponieważ nie podstawiamy bezpośrednio operacji żółwia z innej zmiennej, tylko zapamiętujemy odnośniki do ewaluowanych zmiennych - dzięki temu po zmianie definicji operacji w danej zmiennej, zmienne wykorzystujące jej ewaluację również będą miały inne operacje)
  - stworzenie lub zmiana definicji podstawienia istniejącej zmiennej ('Production')
  - stworzenie pojedynczej operacji żółwia ('TurtleOperationExecution'), obliczenie jej skutów i zapamiętanie na potrzeby drugiej części interpretacji

  - stworzenie ciągu operacji żółwia podstawionych pod daną zmienną ('LiteralExecution'), obliczenie jej skutów i zapamiętanie na potrzeby drugiej części interpretacji (jeśli zmienna ma operacje żółwia, są one po prostu przepisywane, jeśli natomiast posiada ewalucję, następuje rekurencyjne wywołanie - przerywane, gdy natrafimy już na zwykłe operacje, a nie odnośniki do innych zmiennych)

- Variable - przechowuje informacje o jednej zmiennej, to znaczy jej produkcję i (operacje żółwia ALBO zmienne opisujące ewaluację). Variable są przechowywane w VariableMap (mapa: string:Variable).

- TurtleOperation - klasa abstrakcyjna dla różnych rodzajów operacji żółwia, jest to co innego niż TurtleOperation w parserze. TurtleOperation są 'wykonywane' ('apply') w sposób polimorficzny.

- TurtleState, UtmostTurtleCoordinates, DrawingContext - potrzebne przy wykonywaniu operacji żółwia (TurtleState zarówno w pierwszej jak i drugiej fazie interpetacji, UtmostTurtleCoordinates - tylko w pierwszej, do wyznaczenia skrajnych współrzędnych, DrawingContext - tylko przy rysowaniu).

Testowanie

- tests/lexer_tests.cpp - testowanie leksera, sprawdzane są wszystkie rodzaje tokenów w różnych konfuguaracjach, a także błędny kod - testy polegają głównie na sprawdzaniu, czy zwrócony został odpowiedni token, z odpowiednią zawartością

- tests/parser_tests.cpp - testowanie parsera, sprawdzane są wszystkie rodzaje struktur AST w różnych konfuguaracjach, a także błędny kod - testy polegają głównie na **wtórnej generacji kodu na podstawie AST** i porównaniu tego z kodem wejściowym (wynik powinien być taki sam, z dokładnością do białych znaków i komentarzy)

- tests/semantics_tests.cpp - testowanie analizatora semantycznego, nie są to testy w pełni pokrywające wszystkie możliwe jego użycia - w przypadku ostatniego modułu programu ważniejsze były testy funkcjonalne ("czy program działa")

- testowanie funkcjonalne - przykładowe kody testujące są dostępne w katalogu test_code

5 Możliwe zmiany

- Napisanie generycznego leksera i parsera (lub skorzystanie z gotowego generatora), co pozwoliłoby np. na swobodne dodawanie nowych słów kluczowych - operacji żółwia.

- Refaktoryzacja kodu, w szczególności w parserze (sprawdzanie tokenu rozpoczynającego daną strukturę gramatyczną w funkcji parsującej tę strukturę, a nie poza nią).
- Dokładniejsze testy jednostkowe analizatora semantycznego.
- Poprawa obsługi błędów - program kończy się po napotkaniu pierwszego błędu i wypisuje, gdzie on wystąpił. W przypadku niektórych błędów składniowych po wystąpieniu błędu można kontynuować parsowanie, a w przypadku niektórych błędów semantycznych - podawać czytelniejsze komunikaty błędów.