AGH
Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie
Wydział Fizyki i Informatyki Stosowanej

Praca inżynierska

Krystian Wojtas

kierunek studiów: **informatyka stosowana**

kierunek dyplomowania: **metody numeryczne**

Kompilator języków klasy LL do wybranego kodu bajtowego

Opiekun: **dr inż. Maciej Wołoszyn**

Kraków, styczeń 2011

Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie i nie korzystałem ze źródeł innych niż wymienione w pracy.

.....
(czytelny podpis)

Na kolejnych dwóch stronach proszę dołączyć kolejno recenzje pracy popełnione przez Opiekuna oraz Recenzenta (wydrukowane z systemu MISIO i podpisane przez odpowiednio Opiekuna i Recenzenta pracy). Papierową wersję pracy (zawierającą podpisane recenzje) proszę złożyć w dziekanacie celem rejestracji.

Spis treści

1 Wstęp

Celem pracy jest omówienie zasady działania kompilatora języków o gramatykach klasy LL na przykładzie Pascala oraz praktyczne zaimplementowanie algorytmów skanowania tokenów z wykorzystaniem wyrażeń regularnych, parsowania wraz z konstrukcją drzewa rozbioru syntaktycznego oraz generowania wynikowego kodu bajtowego.

1.1 VM

Obecnie oprogramowanie wysokopoziomowe często tworzy się w językach z natury niezależnych od sprzętu oraz systemu operacyjnego, na których jest ono uruchamiane. W jaki sposób jest to realizowane? Otóż kod napisany przez programistę przetwarzany jest w procesie zwanym kompilacją do postaci kodu pośredniego - bajtkodu. Bajtkod jest całkiem bliski asemblerowi, z tą różnicą że jego procesor nie istnieje w rzeczywistości. Dlatego do uruchomienia potrzebny jest w systemie proces maszyny wirtualnej, która iterpretuje kod pośredni i na tej podstawie zadaje procesorowi odpowiednie instrukcje kodu maszynowego. Maszyna wirtualna przejmuje na siebie obowiązki zarządzania dostępnymi rejestrami i pamięcią.

Jak widać VM zapewnia warstwę abstrakcji, która unifikuje różnorodność architektur komputerowych i systemów operacyjnych. To właśnie gwarantuje przenośność pisanego wysokopoziomowego kodu.

1.2 Założenia

Projekt z założenia ma czytać dowolny język, jeśli spełnia on warunki gramatyki klasy LL. Definicja języka zapisana jest w zewnętrznym pliku. Projekt może też generować dowolny bytecode. Aby to osiągnąć i umożliwić wykorzystanie pełnej optymalizacji, nawet dla najbardziej kuriozalnych konstrukcji opcodów, do syntezy wykorzystuje się klasę abstrakcyjną. Zgodnie ze wzorcem projektowym Inverse of Control w czasie przetwarzania źródeł wywołuje się na niej odpowiednie metody. Ukonkretyzowanie tej klasy powoduje generowanie dowolnego kodu pośredniego.

2 Perl

Autorem języka jest Larry Wall, zaczął nad nim pracę w 1987r. Dokładnie 18 grudnia udostępnił pierwsza maszynę wirtualną na grupie dyskusyjnej comp.sources.misc. Przez następne lata język gwałtownie się rozwijał. Zyskał sobie rzesze zwolenników ze względu zarówno na zwinność integracji potężnej mocy drzemiącej w konstrukcjach wyrażeń regularnych jak i łatwość implementacji złożonych struktur danych. Funkcjonalny kod powstaje szybko i przyjemnie. W projekcie używałem wersji

```
$perl -version
This is perl 5, version 12, subversion 2 (v5.12.2) built for i686-linux-multi
```

Rysunek 1: Larry Wall, twórca języka

2.1 Dynamiczna VM

Najciekawszą cechą maszyny wirtualnej Perla jest dynamika. Może ona zmieniać swoje zachowanie w czasie kompilacji po załadowaniu projektowanych ku temu modułów. Przykładowo w Perlu de facto nie występuje konstrukcja switch znana z innych języków programowania

```
switch(liczbaCalkowita) {
  case 0:
    print("Wybrano 0");
    break;
  case 1:
    ..
  default
    ..
}
```

Jednakże po użyciu modułu switch[1] dostępna staje się konstrukcja

```
use feature "switch";
given($literalZnakowy) {
  when(/regExp1/) {
    print("Literal pasuje do wyrazenia 1");
  }
  default {
    print("Zachowanie domyslne");
  }
}
```

Jest to switch z tą różnicą, że `$literalZnakowy` jest badany pod kątem pasujących wyrażeń regularnych, a więc jest znacznie potężniejszy.

2.2 Referencje

Ogromną zaletą jest łatwość tworzenia i operowania złożonymi strukturami danych. Są trzy podstawowe typy zmiennych: skalar będący liczbą całkowita, literałem znakowym bądź wskaźnikiem, oraz tablica i hash.

Mając referencję do struktury `$r`, sprawdzamy na jaki typ wskazuje stosując `ref $r`. Znając jej typ wykonamy rzutowanie na skalar `$$r`, tablicę `@$r` lub na hasha `%$r`. Dla zobrazowania idei posłużę się kilkoma przykładami uruchomionymi w debuggerze.

[1] http://www.misc-perl-info.com/perl-switch.html inne sposoby konstrukcji switcha

```
$ perl -d -e 1

Loading DB routines from perl5db.pl version 1.33
Editor support available.

Enter h or ‘h h’ for help, or ‘man perldebug’ for more help.

main::(-e:1): 1
  DB<1> $liczba = 3;

  DB<2> $literal = ’drugie’.$liczba;

  DB<3> @tab = ( 1, ’dwa’, $liczba, \$literal );

  DB<4> x \@tab
0  ARRAY(0x8edb170)
   0  1
   1  ’dwa’
   2  3
   3  SCALAR(0x8edb130)
      -> ’drugie3’
  DB<5> %hash = ( ’atrybut’ => ’przykladowy’, ’tabP’ => \@tab );

  DB<6> $tab2P = [ [ @tab ], \%hash ];

  DB<7> x $tab2P
0  ARRAY(0x8f11d80)
   0  ARRAY(0x8edb010)
      0  1
      1  ’dwa’
      2  3
      3  SCALAR(0x8edb130)
         -> ’drugie3’
   1  HASH(0x8edb370)
      ’atrybut’ => ’przykladowy’
      ’tabP’ => ARRAY(0x8edb170)
         0  1
         1  ’dwa’
```

```
2  3
3  SCALAR(0x8edb130)
   -> REUSED_ADDRESS
```

W pierwszych 3 krokach tworzone są zmienne, używam konkatenacji (znak .) i dereferencji (znak \). Następnie wyświetlam strukturę na podstawie pobranego jej adresu. Tworzę hash oraz zmienną tab2P, którą drukuję. Widać, że wyrażenie `$tab2P[1]->{tabP}` wskazuje przez adres 0x8edb170 na tablice `\@tab`. Natomiast `$tab2P[0]` wskazuję na identyczną tablicę, jednak znajdującą się pod innym adresem. Kopiowanie wykonaliśmy w kroku 6: `[ @tab ]` utworzyło nową, anonimową tablicę o zawartości takiej jak `@tab` i zwróciło do niej referencję, zapisaną właśnie w `$tab2P[0]`.

2.3 Prawda

Skalar `$zmienna` jest fałszem gdy jest pusty tj. 0, ''. Ta sama maksyma dotyczy tablic i hashy. A co nie jest fałszem, jest prawdą.

2.4 OOP

Mechanizm klas jest dość prymitywny. Procedury grupuje się w pakiety używając słowa package. W tej przestrzeni nazw wywołuje się procedurę pełniącą rolę konstruktora, zazwyczaj new lub create

```
use NowyModul;
$nm = NowyModul->new( 'argument' );
```

Używając strzałki przekazujemy jako pierwszy argument nazwę modułu NowyModul, dopiero kolejnym będzie 'argument'. Zapis jest ekwiwalentny[2]

```
$nm = NowyModul::new('NowyModul', 'argument');
```

Znajduje to zastosowanie w konstruktorze

```
package NowyModul
sub new {
  $class = shift; # pobiera pierwszy argument czyli nazwe modułu
  $self = {};  # tworzy anonimowy hash i przypisuje do niego referencje
  bless $self, $class; # sztuczka obiektu
  $self->{arg} = shift; # wpis w hashu kolejnego argumentu
  return $self;
}
```

[2] my $nm = new NowyModul('argument') również wywiera ten sam efekt

Trik polega na zmianie typu wskaźnika w linii bless. Funkcja ta wiąże w wewnętrznych strukturach vm nazwę pakietu `$class` tj NowyModul ze strukturą `$self`, tu referencja do hasha. Od tej pory `$self` jest wciąż tą samą strukturą, jednak tym razem z przypisaną własną przestrzenią nazw. Można więc wołać na niej funkcje zdefiniowane w jej pakiecie. Programista musi zagwarantować że funkcje te będą poprawnie operować stworzoną strukturą, można już powiedzieć, obiektu.

Czym byłoby OOP bez dziedziczenia? Perl wychodzi naprzeciw potrzebom programistów hierarchizacji klas i unikania redundancji pisanego kodu, oczywiście na swój własny sposób. Jeśli wywoływanej funkcji nie znajdzie w danym pakiecie, sprawdza jego publiczną tablicę `@ISA`. Przegląda kolejne jej wpisy nazw modułów i na nich próbuje odpalić brakującą funkcję. `@ISA` każdego napotkanego modułu także jest sprawdzana. Możliwe jest zatem wielodziedziczenie

```
our @ISA = qw(KlasaBazowa1 KlasaBazowa2);
```

2.5 AOP

Aspekty są paradygmem programowania. W jego terminologii wywoływanie metody nazywane jest punktem złączenia. Dowolny zestaw punktów złączeń, inaczej dowolnie wybrany zestaw metod, nosi nazwę punktu przecięcia. Punktowi przecięcia można zaserwować tzw. radę, tj. wykonanie dodatkowego kodu w momencie powrotu z punktu złączenia, bezpośrednio przed jego nastąpieniem, w przypadku zwrócenia wartości lub wystąpienia wyjątku. Radą uzyskujemy dostęp do kontekstu wywołania metody dzięki czemu można modyfikować przekazane argumenty, zwracać zupełnie inną wartość.

Listing 1: Przykład użycia aspektów

```
1  #!/usr/bin/perl
2  use Aspect;
3
4  sub dodaj
5  {
6    my ( $a, $b ) = @_;
7    return $a + $b;
8  }
9
10 after { # rada 'after'
11   my $a = $_[0]->{params}->[0];
12   my $b = $_[0]->{params}->[1];
13   $_[0]->{return_value} = '5' if($a == 2 and $b == 2);
14 } call qr/dodaj/; # punkt zlaczenia - metoda 'dodaj'
15
16 print '1+1='.dodaj(1, 1)."\n";
17 print '2+2='.dodaj(2, 2)."\n";
```

Wyjście programu:

```
1+1=2
2+2=5
```

Jak to jest zrobione? - chciałoby się rzec przywołując tytuł popularnego programu na kanale Discovery.*hah del* Adres funkcji przed i po zastosowaniem rad sie nie zmienia.

Co ciekawe, porównując języki z platformy .Net lub Jave, tam dodatkowe słowa kluczowe nie mogą zostać użyte i aby osiągnąć AOP należy używać dedykowanych kompilatorów z poziomu źródeł lub ponownie przetwarzać kod bajtowy uzupełniając go o brakującą funkcjonalność w punktach złączeń.

3 Kompilator

W procesie przetwarzania kodu źródłowego programu wyróżniamy główne etapy front, w którym przeprowadza się analizy zadanego kodu, oraz end, w którym produkuje się z zebranych danych kod bajtowy.

3.1 Leksyka

Tekst źródłowy jest serią znaków, należy z nich wydobyć kolejne wzorce jak słowa kluczowe, nazwy, liczby zmienno- lub stałoprzecinkowe, operatory. Należy znalezionym słowom przyporządkować typ i wrzucać je do kolejki.

3.1.1 Terminy

Strumień znaków tekstu źródłowego dzieli się na mniejsze ciągi nazywane leksemami na podstawie znanej klasyfikacji. Klasyfikacja zapewnia przyporządkowanie leksemowi odpowiadającego mu typu - klasy. Każda klasa ma własną nazwę, słowny opis ją wyrażający, reprezentowane wyrażenie regularne i możliwe znaki następujące po leksemie.

3.1.2 Wymagania

Stosuje się warunek znaku występującego bezpośrednio po wyrażeniu. Dzięki temu ciąg returnD nie zostanie trafiony przez RETURN, a złapie go NAZWA, czego oczekujemy.

Liczba zmiennoprzecinkowa zawiera w sobie w początkowych znakach liczbę stałoprzecinkową. Aby ją prawidłowo rozpoznać nadaje się leksemom priorytety według kolejności deklaracji w opisie języka.

Należy dopisać również pewne wzorce, które ewidentnie są błędami w kodach źródłowych. NSTR oraz LERROR są tego praktycznymi dowodami. Takie leksemy ujawnią niesiony błąd w kolejnym kroku analizy podczas budowania drzewa rozbioru.

3.1.3 Opis języka

Klasyfikacja zawarta jest w pliku opisu języka. Interesujące są pierwsza kolumna ze słowami pisanymi wielkimi literami - nazwa klasy oraz trzy kolumny z wyrażeniami zawartymi pomiędzy czwartym separatorem. Pierwsze takie wyrażenie regularne charakteryzuje leksem, następne wyrażenie wyszczególnia możliwe znaki występujące po nim. Ostatnie wyrażenie stanowi słowny opis co klasa wyraża. Przykładowe wiersze, czwarty separator to !

```
WHILE !while! !\s|,|\.|:|;|\(|\)|\+|-|\*|\/|<|>|=! !slowo kluczowe: while!
LOGIKA !true|false! !\s|\)! !wyrazenie logiczne!
LZNAKOWY !"(\w|\s|_|\+|-|\$|~|#|&|@|:)*"! !! !literal znakowy!
NSTR !"[^"\n]*! !\n! !NIEZAMKNIETY STRING!
```

```
NAZWA ![a-zA-z]\w*! !\s|,|\.|:|;|\(|\)|{|}|\+|-|\*|\/|<|>|=! !nazwa wlasna!
LERROR !([^\s]+)! !,|\.|:|;|\(|\)|\+|-|\*|\/|<|>|=! !LEKSYKALNY BLAD!
```

Słowo kluczowe z klasy `WHILE` pasuje tylko do znalezionych ciągów kolejnych małych liter `while`, a następować po nim może którykolwiek ze znaków `spacja`, `tab`, `,`, `.`, `:`, `;`, `(`, `)`, `+`, `-`, `*`, `/`, `<`, `>`, `=`. Klasa `LOGIKA` żąda słowa `true` lub `false`, po nim biały znak. Dopasowując `LZNAKOWY` wymagamy aby zaczynał i kończył się cudzysłowem, w środku mogą wystąpić litery, cyfry i widoczne dopuszczalne znaki.

Zapis wczytywania klasyfikacji z pliku opisu języka

```
if($linia =~
  /^\s*(\w+)[^$t3]*$t3\s*$t4([^$t4]+)$t4\s*$t4([^$t4]*)$t4\s*$t4([^$t4]*)$t4/) {


  push @klasyfikacja, {
    produkcja => $1,
    wzorzec => "($2)",
    terminator => "($3)",
    opis => $4
  };
}
```

3.1.4 Logika

Algorytm wyszukiwania leksemów analizuje kolejne linie kodu użytkownika i próbuje odnaleźć w nich każdy ze zdefiniowanych wzorców. Zapamiętuje wzorzec który zaczyna się od najwcześniejszej pozycji w linii. Jeśli kilka wzorców zacznie się od tego samego miejsca, priorytet ma ten, którego definicja jest wcześniejsza. Po wybraniu leksemu, jest on wycinany z linii wraz z białymi znakami go otaczającymi i trafia do kolejki. Linie są analizowane do wyczerpania posiadanych znaków.

Listing 2: główna część metody analizuj z klasy Kompilator::Pascal::Leksyka

```
1  foreach my $znaki (@zrodlo) {
2    my $ucieteZnaki = 0;
3    my $poczatek;
4    do {
5      $poczatek = -1;
6      $leksem = { slowo => 0, opis => 0, klasyfikacjaNr => 0, linia => 0, poczatek
           => -1};
7      for my $klasa ( @klasyfikacja ) {
8        if($znaki =~ /(\s*$klasa->{wzorzec}$klasa->{terminator})/) {
9          my $p = $-[0];
10         $1 =~ /$klasa->{wzorzec}/;
11         my( $produkcja, $slowo_, $opis_, $poczatek_, $koniec_ ) = ( $klasa->{
               produkcja}, $1, $klasa->{opis}, $p + $-[0], $p + $+[0] );
12         if($poczatek < 0 or $poczatek_ < $poczatek) {
13           $leksem->{produkcja} = $produkcja;
14           $leksem->{slowo} = $slowo_;
15           $leksem->{opis} = $opis_;
16           $leksem->{linia} = $linia;
17           $leksem->{poczatek} = $poczatek_ + $ucieteZnaki;
18           $poczatek = $poczatek_;
19           $koniec = $koniec_;
20         }
21       }
22     }
23     if($poczatek >= 0) {
24       my $ucieteSpacje = length $leksem->{slowo};
25       $leksem->{slowo} =~ s/^\s+//;
26       $ucieteSpacje -= length $leksem->{slowo};
27       $ucieteZnaki += $ucieteSpacje + $koniec;
28       $znaki = substr $znaki, $koniec;
29       #normalnie w tym miejscu
30       push @{$self->_leksemyP}, $leksem;
31       #jednak dla potrzeb debugowania moznaby wywolywac ten fragment w osobnej
             funkcji
32       #dzieki temu mozna sie podpiac do rosnacego stosu aspektem
33       #$self->dolozLeksem( $leksem );
34     }
35   } while($poczatek >= 0);
36   $linia++;
37 }
```

3.2 Syntaktyka

Mamy zestaw słów łącznie z ich typami. Na ich podstawie budowana jest struktura drzewiasta zapisanego programu tzw. drzewo rozbioru syntaktycznego. Efektem ubocznym tej analizy jest zbadanie poprawności użytych słów tzn. wyłapywane są błędy takie jak pozbawiona sensu linia 'var var zmienna var'.

3.2.1 Terminy

Tutaj niechybnie spotykamy się z podstawowym pojęciem produkcji. Oznacza ona odzwierciedlenie symbolu w zestaw symboli precyzyjniejszych. Uwarunkowana jest badanym aktualnie leksemem. Symbole są rozwijane do chwili napotkania symbolu terminalowego, w idealnym świecie, oznaczającego aktualny leksem. Wtedy leksem zostaje zawieszony na drzewie i badany zostaje następny w kolejce.

3.2.2 Gramatyka LL

Gramatyka LL to taka, w której czytamy słowo podczas sprawdzania od lewej do prawej, oraz taka, że możemy podjąć podczas wyprowadzenia lewostronnego jednoznaczną decyzję, którą produkcje wybrać.

Gramatyka LL(1) to gramatyka LL taka, że podczas podejmowania decyzji, którą produkcję wybrać analizujemy dokładnie jeden symbol słowa.

Dla zobrazowania podam przykład praktyczny z języka C:

```
int a;
int b();
```

W pierwszej instrukcji zdefiniowaliśmy zmienną `a`, w drugiej funkcję `b`. Zatem po napotkaniu słowa `int` nie można stwierdzić, czy czytana instrukcja będzie deklaracją zmiennej czy funkcji. Inaczej jest w języku Pascal

```
var a:integer;
function b(): integer;
```

W tym przypadku wiemy po pierwszym słowie co dokładnie będzie deklarowane i jakich kolejnych leksemów można się spodziewać. W Pascalu także miejsca deklaracji są ściśle określone. Po drugiej stronie, w C, można deklarować zmienne na każdym poziomie zagnieżdżania bloków.

Mówimy, że parser LL wykonuje parsowanie metodą zstępującą (ang top-dawn) ponieważ z produkcji najwyższego poziomu próbuje dotrzeć do liści.

3.2.3 Wymagania

Celem tego etapu jest stworzenie drzewa rozbioru syntaktycznego. Korzeniem drzewa jest byt abstrakcyjny `program` - nieterminal. Kolejnymi gałęziami tego symbolu są

```
deklaracjeOpcj proceduryOpcj START instrukcje END
```

Czy gałąź `deklaracjeOpcj` się rozwinie zależy od przetwarzanego kodu źródłowego. Jeśli zaczyna on się od słowa `proc` to widać, że użytkownik nie deklaruje żadnych zmiennych globalnych, podczepiony tutaj zostanie terminal abstrakcyjny `EPSILON` i rozkład `deklaracjeOpcj` się kończy. Z kolei dla `proceduryOpcj` słowo źródłowe `proc` jest jak najbardziej oczekiwanym słowem i nastąpi rozwój tej gałęzi. Po jego zakończeniu musi wystąpić terminal `START`, czyli słowo kluczowe `start` będzie musiało na tym etapie wystąpić w źródle, inaczej zgłaszany jest błąd kompilacji.

Aby poznać dokładne działanie algorytmu zapoznajmy się z danymi wejściowymi. Są to kolejka leksemów pozyskana w poprzedniej analizie oraz wykreowana na bieżące potrzeby tablica parsingu. Tablica parsingu przyporządkowuje każdej parze aktualnie analizowanego nieterminala i aktualnie badanego leksemu z kolejki źródła rozwinięcie w szereg symboli precyzyjniejszych. Być może będzie to już terminal (i być może `ERR`).

3.2.4 Opis języka

Ważne są kolumny pierwsza z nazwa klasy wyrażenia, druga - rozwinięcie i trzecia - tablica parsingu. Klasy pisane wielkimi literami to symbole terminalowe, reszta to nieterminale.

3.2.5 Logika

Rozważmy najprostszy program źródłowy, który nie robi nic prócz deklaracji zmiennej globalnej

```
var zm1:float;
start
end
```

Jak podałem wcześniej, zaczynamy od nieterminala `program`, wskaźnik do tablicy z tym słowem ładowany jest na stos.

```
@stosy = (
  [ 'program' ],
  );
```

W głównej pętli adres tej tablicy natychmiast jest ściągany jako że leży na szczycie. Pozyskiwane jest pierwsze (i jedyne) słowo z tejże tablicy, oczywiście `program`, i zapisywane jest jako `$aktualnaProdukcja`. Nie jest to w żadnym wypadku terminal, zatem poddajemy je obróbce tablicy parsingu. Jeśli pierwszy leksem z kolejki nie może rozpoczynać kodu źródłowego, uzyskamy terminal `ERR`. `Var` może, więc dostajemy rozwinięcie w tablicę symboli

```
program -> deklaracjeOpcj proceduryOpcj START instrukcje END
```

Jej adres ładowany jest na stos, a wcześniej ładowany jest adres tablicy zawierającej `program`, która już jest pusta.

```
@stosy = (
  [],
  [ 'deklaracjeOpcj', 'proceduryOpcj' 'START' 'instrukcje' 'END' ],
  );
```

Do drzewa rozbioru trafia pierwsze dziecko `program`, a skoro pierwsze to jest to korzeń.

Kolejna iteracja pętli. Tym razem aktualną produkcją będzie `deklaracjeOpcj`, wciąż nieterminal. Parsing od wektora (`deklaracjeOpcj`, `var`) zwróci rozwinięcie

```
deklaracjeOpcj -> deklaracja deklaracjeOpcj
```

Zrzut stosu:

```
@stosy = (
  [],
  [ 'proceduryOpcj' 'START' 'instrukcje' 'END' ],
  [ 'deklaracja', 'deklaracjeOpcj' ],
  );
```

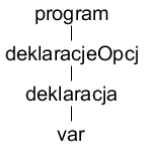
`deklaracjeOpcj` zawieszany jest na drzewie jako pierwszy potomek `program` Następna iteracja. Przez analogię

```
deklaracja -> VAR nazwy DWUKROPEK zmD_typ SREDNIK
```

```
@stosy = (
  [],
  [ 'proceduryOpcj' 'START' 'instrukcje' 'END' ],
  [ 'deklaracjeOpcj' ],
  [ 'VAR' 'nazwy' 'DWUKROPEK' 'zmD_typ' 'SREDNIK' ],
  );
```

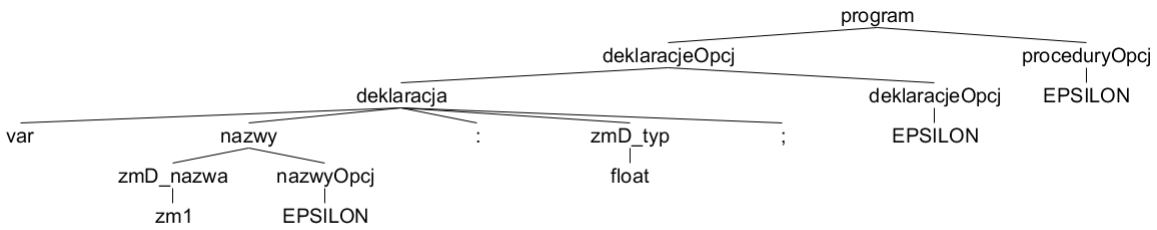
Na czwartym poziomie wreszcie osiągnęliśmy produkcję terminalową `VAR`. Badany leksem jest dokładnie tej samej klasy. Mamy zgodność typów, zatem wyrzucamy leksem z kolejki, zawieszamy go na drzewie i badamy kolejny, zaczynając porównywać go z pierwszym słowem na najwyższym stosie, będą to `nazwy`

```
@stosy = (
  [],
  [ 'proceduryOpcj' 'START' 'instrukcje' 'END' ],
  [ 'deklaracjeOpcj' ],
  [ 'nazwy' 'DWUKROPEK' 'zmD_typ' 'SREDNIK' ],
  );
```



Rysunek 2: Pierwszy terminal VAR

Myślę że sam algorytm zaczyna się już klarować. Dlatego pominę dalszy rozwój `deklaracja` i przejdźmy do chwili jej zakończenia. Rysuję drzewo rozbioru



Rysunek 3: Moment zamknięcia deklaracji

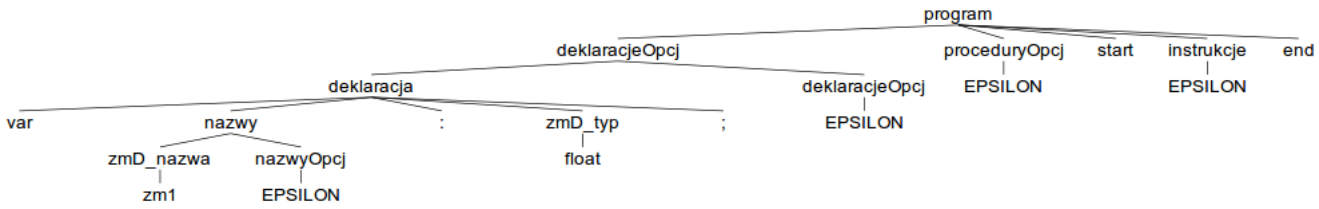
oraz załączam zrzut stosu

```
@stosy = (
  [],
  [ 'proceduryOpcj' 'START' 'instrukcje' 'END' ],
  [ 'deklaracjeOpcj' ],
  );
```

Prawie Dajavu z drugiego poziomu, zniknęła tylko rozpracowana właśnie `deklaracja`. Znów napotykamy `deklaracjeOpcj`, które, jeśli zadeklarowalibyśmy następną zmienną, ponownie pozostawią na stosie | `deklaracja`, `deklaracjaOpcj` | i po rozłożeniu których znowu zastalibyśmy stos w stanie przedstawionym wyżej. Jednakże nic już nie deklarujemy, a badany obecnie leksem jest klasy terminalowej `START`. Parsing dla takich argumentów oddaje `EPSILON`. Cechą terminali jest, że ich gramatyka oddaje siebie samych. Tak więc

```
@stosy = (
  [],
  [ 'proceduryOpcj' 'START' 'instrukcje' 'END' ],
  [ 'EPSILON' ],
  );
```

Po zawieszeniu na drzewie `EPSILON` najwyższy stos będzie pusty. W takiej sytuacji cofamy się na drzewie rozbioru do przodka. `proceduryOpcj` podłączane są do symbolu pierwotnego `program`, do nich `EPSILON`. `START` występuje w leksemie, `instrukcje` są puste i `END` kończy kod źródłowy. Zastając puste stosy, analiza zostaje zakończona z rezultatem:



Rysunek 4: Rozbiór kodu źródłowego

Listing 3: główna część metody analizuj z klasy Kompilator::Pascal::Syntaktyka

```
while(@stosy) {
  $aktualnyStosP = pop(@stosy);
  if(not @{$aktualnyStosP}) {
    $drzewo->parent();
  }
  else {
    $aktualnaProdukcja = shift( @{$aktualnyStosP} );

    given($aktualnaProdukcja) {
      when (/ERR/) {
        if($panicMode == 0) {
          #warning::warnif ?
          printf "BLAD, dla produkcji $poprzedniaProdukcja znaleziono
              niespodziewane slowo '$leksem->{slowo}', nr typu $leksem->{
              klasyfikacjaNr}, linia $leksem->{linia}, poczatek $leksem->{
              poczatek}\n";
        }
        else {
          printf "BLAD, brakuje produkcji $poprzedniaProdukcja w linii $leksem
              ->{linia}\n";
        }
        $panicMode = 1;
        $err = 1;
        $drzewo->parent();
      }
      #TODO del, for debug purpose only
      when (/deklaracja$/) {
        continue;
      }
      when (/^[A-Z]/) {
        if(/EPSILON/) {
          my $produkcjaEpsilon = $drzewo->zrobEpsilon($leksem);
          $drzewo->textNode($leksem);
        }
        else {
          $drzewo->textNode($leksem);
          $leksem = shift(@leksemy);
        }
        push @stosy, $aktualnyStosP;
        $panicMode = 0;
      }
      default {
        push @stosy, $aktualnyStosP;
        $drzewo->appendChild($aktualnaProdukcja);
```

```
41          die "Bledna tablica parsingu\nNie zdefiniowano $leksem->{slowo} z klasy
                $leksem->{produkcja} dla produkcji $aktualnaProdukcja\n"
42            if not defined $parsing{ $aktualnaProdukcja }->{ $leksem->{produkcja}
                  };
43          #wyrazenie rozwija w tablice symboli, rzucana na stosy, nastepna
                produkcje pozyskana ze skrzyzowania warunkow obecnej produkcji i
                badanego leksemu
44          push @stosy, [ @{ $parsing{ $aktualnaProdukcja }->{ $leksem->{produkcja}
                } } ];
45          $poprzedniaProdukcja = $aktualnaProdukcja;
46        }
47      }
48    }
49 }
50 if($err) {
51    printf "ERRORY\n";
52    die "Wystapily bledy podczas analizy syntaktycznej\n";
53 }
```

3.3 Semantyka

Docierając do tego etapu dysponujemy poprawnym pod kątem syntaktycznym drzewem rozbioru. Teraz należy wyodrębnić padające deklaracje nazw własnych i je zapamiętywać wraz z ich kontekstem. W przypadku późniejszego odwołania do takowych sprawdzamy czy nazwa jest nam znana i wyrzucamy błąd, jeśli nie jest.

Semantyka jest zbiorem reguł ograniczających swawolę programisty, jednocześnie gwarantujących przewidywalność działania programu.

3.3.1 Wymagania

Nie mogą istnieć deklaracje zmiennych o tych samych nazwach na tym samym poziomie. Na różnych są dopuszczalne, wtedy deklarowane później przesłaniają wcześniejsze, do których tracimy dostęp.

Wyróżniamy trzy rodzaje wartości: wartości stałe oraz zapisane w zmiennych globalnych i lokalnych. Typ może być prosty: literał znakowy, liczba całkowita, zmiennoprzecinkowa, wartość logiczna; lub być typem tablicowym, czyli zawierać zestaw typów prostych. W takim wypadku przed użyciem należy zaalokować dla niego miejsce i nie wykraczać poza nie. Nie można drugi raz alokować miejsca dla zmiennej już zaalokowanej, nie można dwa razy tego samego miejsca zwolnić. Zapamiętywane własności operandu stanowią jego maskę.

W definicjach funkcji wolno wywoływać funkcje, których deklaracja nie jest jeszcze znana, a nastąpi w dalszej części pliku. Wymusza to dwuprzebiegowy proces analizy - wstępny zapamięta wszystkie deklaracje funkcji (wraz z maskami przyjmowanych argumentów), kolejny zajmie się właściwą pracą.

Wywołania funkcji można zagnieżdżać. Oznacza to, że wywołując funkcję można jako argumenty stosować wyrażenie i przekazywać jego końcowy wynik. A w skład wyrażenia może wychodzić wywołanie kolejnej funkcji również ze zbiorem wyrażeń w argumentach.

Dla procesu syntezy stracą znaczenie nazwy zmiennych i procedur, ważny stanie się przyporządkowany im numer i maska. Dlatego nazwy własne pozostaną w obszarze semantyki, która pozostawi je sobie samej.

1dd	stała
2dd	zmienna globalna
3dd	zmienna lokalna

Tabela 1: Rodzaje wartości

d0d	typ prosty
d1d	typ złożony, niezaalokowany
d2d	typ złożony, zaalokowany

Tabela 2: Tablice

dd1	integer
dd2	float
dd3	boolean
dd4	string

Tabela 3: Typy wartości

Tabela 4: Rozkład masek operandów

3.3.2 Opis języka

Pod uwagę brane są linie zawierające piąty separator. Po nim następuje wyrażenie stanowiące pełnioną przez klasę funkcję. Para nazwa klasy i funkcja jest opisem semantyki. Funkcje:

procedura	Znacznik informuje o wejściu do ciała procedury
prD_nazwa	Następująca nazwa własna będzie nazwą analizowanej właśnie procedury
prD_typ	Następujący typ będzie typem analizowanej właśnie procedury
parametryOpcj	Następują deklaracje parametrów procedury
parametr	Następuje deklaracja parametru procedury
deklaracja	Deklarowana będzie zmienna lub parametr procedury
zmD_nazwa	Następująca nazwa własna będzie nazwą deklarowanej zmiennej
zmD_typ	Następujący typ będzie typem deklarowanej zmiennej
prW_nazwa	Następująca nazwa własna będzie nazwą wywoływanej procedury
pmW	Następuje przekazywanie argumentu do wywoływanej procedury
zmW_nazwa	Następująca nazwa własna będzie nazwą wywoływanej zmiennej
zmA_nazwa	Następująca nazwa własna będzie nazwą zmiennej alokowanej
zmZ_nazwa	Następująca nazwa własna będzie nazwą zmiennej zwalnianej
alokacjaD	Znacznik informuje o alokacji pamięci
zwolnienieD	Znacznik informuje o zwalnianiu zajętej pamięci
liczbaint	Analizowany typ jest liczbą całkowitą
liczbaf	Analizowany typ jest liczbą zmiennoprzecinkową
logika	Analizowany typ jest wartością logiczną
lznakowy	Analizowany typ jest literałem znakowym
arrayof	Analizowany typ jest złożony
lubwyrazenie	Znacznik informuję o następującym nowym wyrażeniu
przypisanieD	Znacznik informuje o mającym nastąpić przypisaniu wartości do zmiennej
plusm1	Następujący operator jest jednoargumentowym +-
plusm razyd zaleznosc and or rowne nawoo nawoz naklz	Następujący operator dwuargumentowy
blokD	Deklaracja bloku. Wewnątrzne deklaracje zmiennych przesłonią obowiązujące
zwrot	Następuje powrót z procedury

3.3.3 Logika

Działanie polega na rekursywnym przejściu przez drzewo rozbioru i śledzeniu aktywności gałęzi funkcyjnych. Dotarłszy do znaczącego terminatora sprawdzamy ustawienia zworek aktywności i w zależności od ich położeń wykonujemy akcje na obiekcie syntezy.

Funkcja _szukajFunkcji zbada tym sposobem definicje procedur - zapamięta maski parametrów oraz nazwę i typ samej procedury. Jest to analiza wstępna.

Listing 4: metoda szukajFunkcji z klasy Kompilator::Pascal::Semantyka

```
1  sub _szukajFunkcji
2  {
3    my ( $self, $wezel ) = @_;
4    #TODO deklaracja pozniej
5    my $produkcjaN = 0;
6    foreach my $wezelN (@{$wezel->{dzieci}})
7    {
8      $produkcjaN = $wezelN->{produkcja};
9      #wlaczanie zworek
10     if($prDA == 0) {
11       if($produkcjaN =~ /^$prD$/) {
12         $prDA = 1;
13       }
14     }
15     else {
16       if($pmDA == 0) {
17         if($produkcjaN =~ /^$pmD$/) {
18           $pmDA = 1;
19         }
20         elsif($produkcjaN =~ /^$prD_nazwa$/) {
21           $prD_nazwaA = 1;
22         }
23         elsif($produkcjaN =~ /^$prD_typ$/) {
24           $prD_typA = 1;
25         }
26       }
27       else {
28         if($produkcjaN =~ /^$zmD_nazwa$/) {
29           $zmD_nazwaA = 1;
30         }
31         elsif($produkcjaN =~ /^$zmD_typ$/) {
32           $zmD_typA = 1;
33         }
34         elsif($produkcjaN =~ /^$ar$/) {
35           $arAA = 1;
36         }
37       }
```

```
38      }
39
40      #rzezba
41      if($produkcjaN !~ /^0$/) {
42        $self->_szukajFunkcji($wezelN);
43      }
44      else {
45        if($prDA) {
46          if(not $pmDA) {
47            if($prD_nazwaA) {
48              my $nazwa = $wezelN->{leksem}->{slowo};
49              $self->_syntezaP->deklProcNazwaD( $wezelN->{leksem}->{slowo} );
50            }
51            elsif($prD_typA) {
52              $self->_syntezaP->deklProcMaskaD( $produkcjaTypuNr{ uc $wezelN->{
                    leksem}->{slowo} } );
53              $arAA = 0;
54            }
55          }
56          else {
57            if($zmD_nazwaA) {
58              $self->_syntezaP->deklParamNazwaD( $wezelN->{leksem}->{slowo} );
59            }
60            elsif($zmD_typA) {
61              $self->_syntezaP->deklParamMaskaD ( '2'.(($arAA) ? '1' : '0').
                    $produkcjaTypuNr{ uc $wezelN->{leksem}->{produkcja} } );
62              $arAA = 0;
63            }
64          }
65        }
66      }
67
68      #wylaczanie zworek
69      if($produkcjaN !~ /^0$/) {
70        if($prDA == 1) {
71          if($produkcjaN =~ /^$prD$/) {
72            $prDA = 0;
73            $self->_syntezaP->deklProcD();
74          }
75          if($pmDA == 0) {
76            if($produkcjaN =~ /^$prD_nazwa$/) {
77              $prD_nazwaA = 0;
78            }
79            if($produkcjaN =~ /^$prD_typ$/) {
80              $prD_typA = 0;
81            }
```

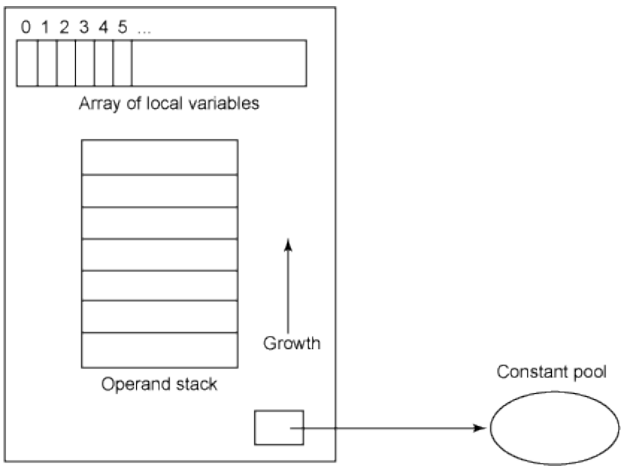
```
82          }
83          else {
84            if($produkcjaN =~ /^$pmD$/) {
85              $pmDA = 0;
86               $self->_syntezaP->deklParamD();
87            }
88            elsif($produkcjaN =~ /^$zmD_nazwa$/) {
89              $zmD_nazwaA = 0;
90            }
91            elsif($produkcjaN =~ /^$zmD_typ$/) {
92              $zmD_typA = 0;
93            }
94          }
95        }
96      }
97    }
98  }
```

Po tym nastąpi właściwa analiza wywołań zmiennych i fukcji w metodzie _sprawdzWywolania. Schemat jej działania jest bardzo podobny.

3.4 Synteza

3.4.1 JVM

Wirtualna maszyna Javy jest maszyną stosową. Każdy wykonywany wątek ma swój osobny stos tzw. ramek. Wywołując metodę, odkładamy na stos nową ramkę. Składa się ona z referencji do pól statycznych klasy wywoływanej metody, tablicy parametrów lokalnych o znanej wielkości (policzonej na etapie kompilacji metody) oraz stosu operandów. Jeśli metoda jest wirtualna, pierwszym parametrem lokalnym jest referencja do `this`.



Rysunek 5: Ramka

Wykonywana metoda działa wg schematu zapisanego we własnym kodzie bajtowym. Może ładować swoje parametry na stos, tam konwertować ich typy, wykonywać obliczenia i zapisywać wynik znowu w parametrach lokalnych. W trakcie działania wywołuje inne metody, wtedy nowa ramka przesłani chwilowo obecną. Po powrocie na stosie bieżącej ramki zastaniemy ewentualny zwrócony operand.

Dane na stosach operandów mają podstawową szerokość 32 bitów. Ładując typ mniejszy, jest on rozszerzany, operując `long`-iem wykorzystujemy dwa słowa.

3.4.2 ONP

Odwrotna Notacja Polska jest sposobem zapisu wyrażeń matematycznych. Opracował ją Charles Homblin opierając się na Notacji Polskiej Jana Łukasiewicza i ją "odwracając". Zapis

```
2 + 3 * 4 - 5
```

W ONP wygląda

```
2 3 4 * + 5 -
```

Do obliczenie wartości wyrażenia pobiera się kolejne symbole z ciągu i w razie napotkania operatora wylicza się nim dwie ostatnie wartości i na ich miejsce wstawiany jest wynik. Kolejne kroki

```
2 3 4 * + 5 -
2 12 + 5 -
14 5 -
9
```

Czemu to służy? Procesor RISC w jednej instrukcji może obliczać dwie wartości ulokowane w dwóch rejestrach, co przenosi się na JVM, która w jednym opkodzie może obliczać dwie wartości ze szczytu stosu. Dlatego złożone wyrażenia należy rozbić na zestaw prostych instrukcji dwuargumentowych. Rozpracowując takie wyrażenie w naturalnym zapisie musimy uważać na priorytety operatorów tzn. wstrzymać się z obliczeniami do momentu uzyskania wyniku operandów o wyższym priorytecie. ONP znakomicie rozstrzyga problem margnizując priorytety w określaniu kolejności operacji.

3.4.3 Kod bajtowy

Spójrzmy na przykładowe opkody

Mnemonik	Opkod	Dod. bajty	stos operandów	opis
iconst_m1	02		-> -1	ładuje stała -1 jako int
iconst_0	03		-> 0	ładuje stała 0 jako int
fconst_1	0b		-> 0.0f	ładuje stałą -1.0 jako float
ldc	12	idx	-> w	ładuje stałą o indeksie `idx`
iload	15	idx	-> w	ładuje parametr `idx` jako int
iload_0	1a		-> w	ładuje parametr 0 jako int
fload_1	23	idx	-> w	ładuje parametr 1 jako float

istore	36	i	w ->	wrzuca wartość inta do parametru `i`
iastore	4f		ref idx w ->	wrzuca wartość inta `w` do tablicy `ref` pod `idx`
aastore	53		ref1 i ref2 ->	wrzuca referencję `ref2` do tablicy `ref1` pod `i`
getfield	b4	idx1, idx2	ref -> w	ładuje pole `idx1<<8+idx2` obiektu ref
putfield	b5	idx1, idx2	ref w ->	zapisuje pole `idx1<<8+idx2` obiektu ref
anewarray	b5	i1, i2	r -> ref	nowa tablica klasy `i1<<8+i2` o rozmiarze `r`
l2i	88		w -> wynik	rzutowanie long na int
i2f	86		w -> wynik	konwersja int do float
fadd	62		w1, w2 -> wynik	dodaje 2 ostanie liczby jako float
iand	7e		w1, w2 -> wynik	logiczny and między 2 intami
fneg	76		w -> wynik	neguje floata
if_acmpeq	a5	b1 b2	ref1 ref2 ->	skok o `b1<<8+b2` jeśli referencje się zgadzają
if_icmplt	a1	b1 b2	w1 w2 ->	skok o `b1<<8+b2` jeśli `w1` mniejsze od `w2`
goto	a7	b1 b2	->	skok o `b1<<8+b2`
ireturn	ac		w -> [pusty]	zwraca inta z metody
invSpec	b7	i1, i2	ref, [arg] -> w	wywołuje metodę `i1<<8+i2` obiektu ref z `arg`
instOf	c1	i1, i2	ref, -> w	sprawdza czy `ref` jest klasy `i1<<8+i2`
athrow	bf	i1, i2	ref -> [pusty] ref	rzuca wyjątek `ref`, czyści stos

Tabela 6: Wybrane opkody

Mnemoniczny jednoliterowy przedrostek uwidacznia typ prymitywny na którym opkod operuje.

Wszystkie opkody zapisane są w jednym bajcie. Pewne z nich wymagają dodatkowych bajtów następujących w strumieniu kodu pośredniego. Opkod `iload` załaduje na stos wartość parametru ramki znajdującego się pod indeksem określonym w następnym bajcie. Istnieją skrócone opkody np zawierające już indeks parametru jak `iload_0`.

Pewne opkody odwołują się do metod lub pól klasy poprzez numer indeksowy zapisany w strumieniu. Wyraża on faktyczne nazwy - przyporządkowanie zapisane jest w plikach .class.

`$ cat A.java`	`$ javap -c A`
`class A {`	`Compiled from "A.java"`
	`class A extends java.lang.Object{`
	`A();`
	`  Code:`
	`   0: aload_0`
	`   1: invokespecial`
	`#1; //Method java/lang/Object."<init>":()V`
	`   4: return`
`  void a() {`	`void a();`
`    int zm1 = 3;`	`  Code:`
`  }`	`   0: iconst_3`
	`   1: istore_1`
	`   2: return`
`  long b(long zm1, float zm2) {`	`long b(long, float);`
`    int zm3 = -1;`	`  Code:`
`    zm2 = zm1 + zm3 * (8 - 2);`	`   0: iconst_m1`
`    return zm1;`	`   1: istore 4`
`  }`	`   3: lload_1`
	`   4: iload 4`
	`   6: bipush 6`
	`   8: imul`
	`   9: i2l`
	`   10: ladd`
	`   11: l2f`
	`   12: fstore_3`
	`   13: lload_1`
	`   14: lreturn`
`}`	`}`

Tabela 7: Porównanie Javy i jej bajtkodu

3.4.4 Konkretyzacja

Zadaniem kompilatora jest generowanie dowolnego bajtkodu poprzez zaimplementowanie pod jego kątem klasy abstrakcyjnej. Aby spełniał walory użytkowe konkretyzacja musi być prosta. Klasa abstrakcyjna:

Listing 5: klasa abstrakcyjna Kompilator::Api::Synteza::Class

```
#!/usr/bin/perl
package Kompilator::Api::Synteza::Class;
use strict;
use warnings;

sub new
{
  my $class = shift;
  my $self = {};
  bless $self, $class;

  $self->_wyjsciePlik( shift );
  $self->_wyjscieP( [] );

  return $self;
}

sub _wyjsciePlik { $_[0]->{wyjsciePlik}=$_[1] if defined $_[1]; $_[0]->{
    wyjsciePlik} }
sub _wyjscieP { $_[0]->{wyjscieP}=$_[1] if defined $_[1]; $_[0]->{wyjscieP} }

sub ldc {}
sub iload {}
sub fload {}
sub bload {}
sub aload {}
sub iaload {}
sub putfield {}
sub getfield {}
sub fstore {}
sub bstore {}
sub astore {}
sub iastore {}
sub iadd {}
sub fadd {}
sub isub {}
sub fsub {}
sub imul {}
sub fmul {}
sub idiv {}
sub fdiv {}
sub faload {}
sub baload {}
sub aaload {}
sub f2i {}
```

```
45 sub i2f {}
46 sub i2b {}
47 sub b2i {}
48 sub newarrayInt {}
49 sub newarrayFloat {}
50 sub newarrayBoolean {}
51 sub newarrayString {}
52 sub invoke {}
53
54
55 sub drukujKod
56 {
57   my $self = shift;
58   my $kody = shift;
59   push @{$self->_wyjscieP}, "$kody\n";
60   return $self;
61 }
62
63 sub zapisz
64 {
65   my $self = shift;
66   open(KLASA, '>', $self->_wyjsciePlik ) or die "Cannot write to file $self->
       _wyjsciePlik, exit $!";
67   foreach my $linia ( @{$self->_wyjscieP} ) {
68     print KLASA $linia;
69   }
70   close(KLASA);
71 }
72
73
74 1;
```

Javowa konkretyzacja:

Listing 6: fragment Kompilator::Java::Synteza::Class

```
sub invoke
{
  my ( $self, $prWNr ) = @_;
  $self->drukujKod( "invoke $prWNr" );
  return $self;
}

sub ldc
{
  my ( $self, $pozycja ) = @_;
  $self->drukujKod("ldc $pozycja");
  return $self;
}

sub getfield
{
  my ( $self, $pozycja ) = @_;
  $self->drukujKod("getfield $pozycja");
  return $self;
}

sub astore
{
  my ( $self, $pozycja ) = @_;
  $self->drukujKod("astore $pozycja");
  return $self;
}

sub iadd
{
  my $self  = shift;
  $self->drukujKod('iadd');
  return $self;
}

sub b2i
{
  my $self  = shift;
  $self->drukujKod('b2i');
  return $self;
}
```

4 Projekt

4.1 Budowa

Projekt zbudowany jest jako moduł przy wykorzystaniu narzędzia Module::Starter dostępnego w bibliotece cpan [3]. Szkielet modułu zawiera plik Makefile.PL, który po uruchomieniu tworzy właściwy Makefile. Mamy w nim dostępne cele takie jak make, make test, make install, make clean. Nie mając uprawnień super użytkownika, nie uda się makefilem modułu osadzić w systemie, czyli przekopiować plików do ścieżek wymienionych w @INCLUDE

```
perl -e "print qq(@INC)"
```

Inne wykorzystane moduły

constant : Deklaracje stałych

strict : Wymusza bardziej restrykcyjne zachowanie VM

warnings : Drukuje ostrzeżenia

warnings::register : Warunkowe ostrzeżenia

Carp : Drukuje diagnostyke lub przerywa program

module "switch" : Konstrukcja "switch"

XML::DOM : Dostęp DOM do drzewa XML

Data::Dumper : Zrzuty struktur

Aspect : Aspektowy paradygm programowania

FindBin : Lokalizacja skryptu

lib : Ścieżki wyszukiwań modułów

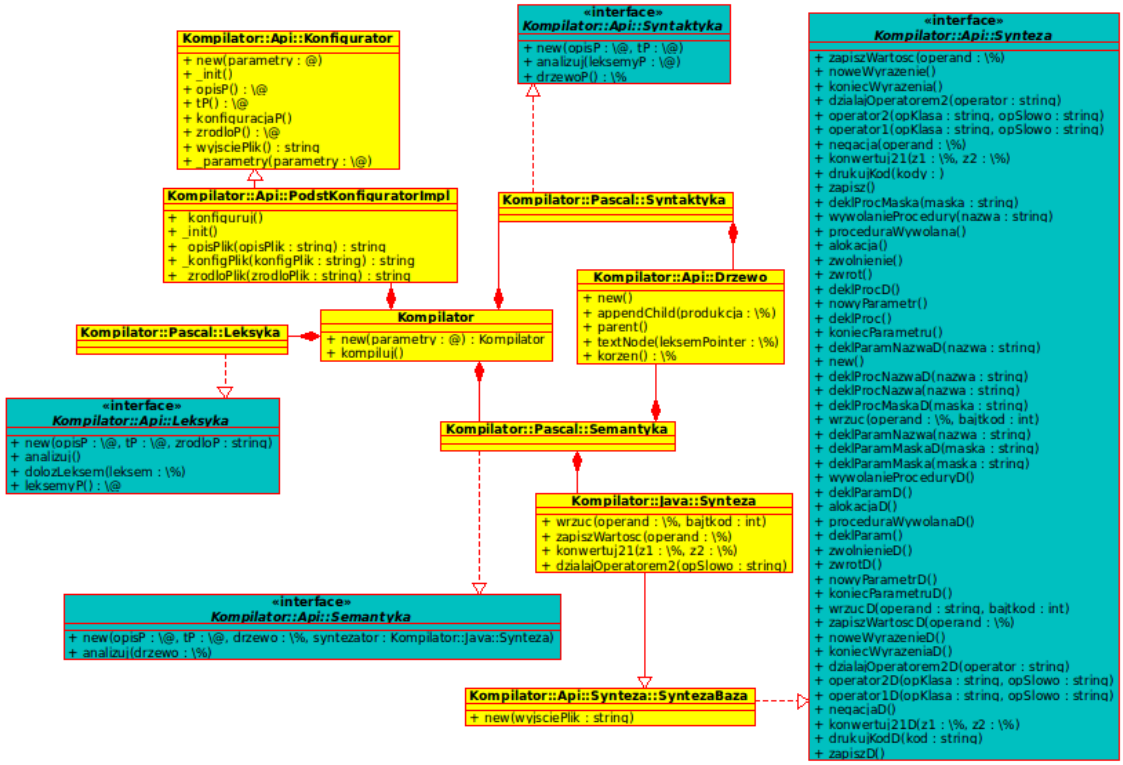
Test::More : Testy

Test::Exception : Testy wyjątków

[3] instalacje w unixach jako root: cpan -i Wybrany::Modul

4.2 Model

4.2.1 Klasy



Rysunek 6: Rdzeń analiz

Kompilator : buduje świat i rozpoczyna analizy

Kompilator::Api::Konfigurator : Baza dla zaimplementowania konfiguracji działania kompilatora

Kompilator::Api::PodstKonfiguratorImpl : Czyta podstawowe pliki wejściowe, ustanawia konfigurację

Kompilator::Pascal::Leksyka : Odpowiada za analizę leksykalną.

Kompilator::Pascal::Syntaktyka : Odpowiada za budowę drzewa rozbioru.

Kompilator::Api::Drzewo : Struktura drzewa rozbioru.

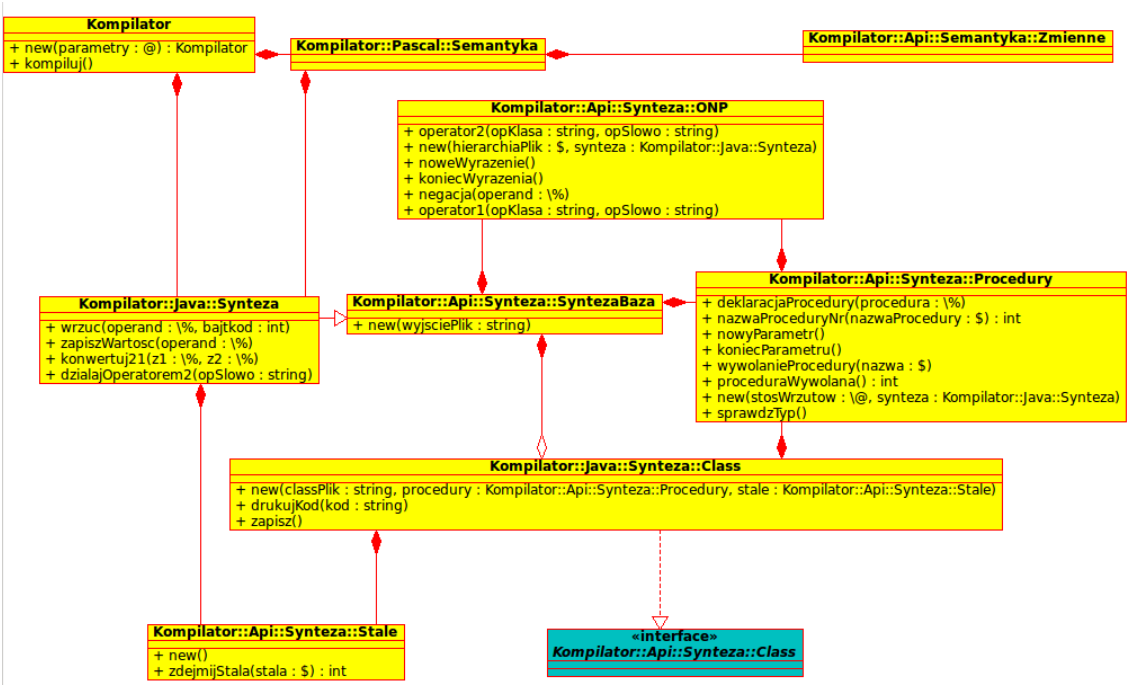
Kompilator::Pascal::Semantyka : Implementuje dwuprzebiegową analizę semantyczną.

Kompilator::Api::Synteza : Interfejs wywołań zwrotnych.

Kompilator::Api::Synteza::SyntezaBaza : Szkielet do syntezy kodu bajtowego. Deleguje wywołania zwrotne do klas specjalistycznych.

Kompilator::Java::Synteza : Dziedziczy po SyntezaBaza, może wpłynąć na zachowanie przodka.

Uproszczenie konkretyzacji wyjściowych bajtkodów jest zrealizowane poprzez zestaw klas



Rysunek 7: End

Kompilator : buduje świat i rozpoczyna analizy

Kompilator::Pascal::Semantyka : Implementuje dwuprzebiegową analizę semantyczną.

Kompilator::Api::Semantyka::Zmienne : Klasa pomocnicza, opiekuje się nazwami zmiennych, przeprowadza deklaracje, sprawdza wywołania. Przechowuje tablicę globalną zmiennych oraz stos lokalnych. Zajmuje się maskowaniem alokacji i dealokacji.

Kompilator::Api::Synteza::SyntezaBaza : Deleguje aktywność gałęzi funkcjonalnych na specjalizowane klasy pomocnicze.

Kompilator::Api::Synteza::ONP : Przetwarza złożone wyrażenia do trójwersowych. Buduje do tego stosy zmiennych i operatorów.

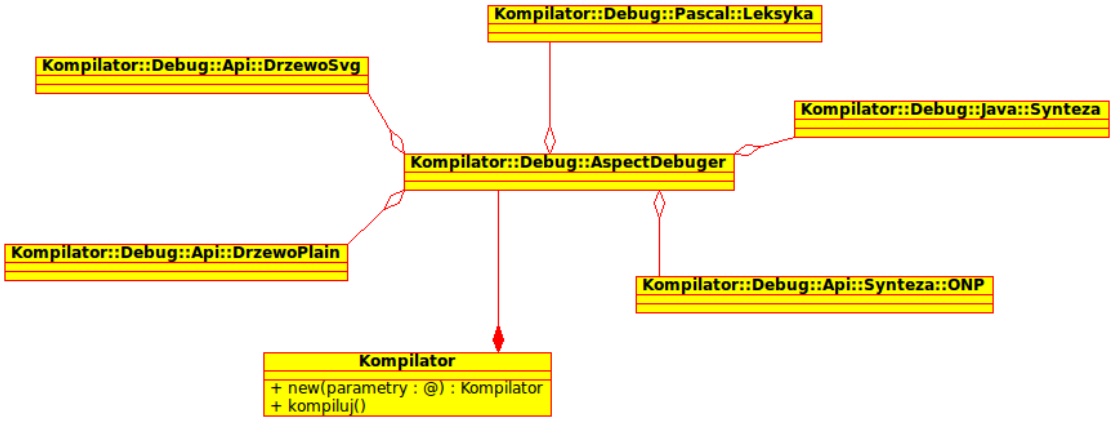
Kompilator::Api::Synteza::Procedury : Przechowuje padające definicje procedur, buduje stos wywołań funkcji. Konwertuje typy wywoływanych argumentów. Wyrzuca błąd jeśli wywołana argumentów nie zgadza się ze zdefiniowaną. Potrzebuje referencji onp w celu obsługi złożonych wyrażeń w wywołaniach.

Kompilator::Java::Synteza : Dziedziczy po SyntezaBaza, może wpłynąć na zachowanie przodka. Operuje maskami w celu wydelegowania właściwego opkodu.

Kompilator::Api::Synteza::Class : Interfejs do implementacji dla konkretnych bajtkodów.

Kompilator::Java::Synteza::Class : Właściwa konkretyzacja produkowanego bajtkodu.

Aspektowo zorienowane logowanie działań poszczególnych etapów



Rysunek 8: Logowanie

Kompilator::Debug::AspectDebugger : Rdzeń logowania aspektowego. Na podstwie otrzymanej konfiguracji tworzy obiekty logowań i uruchamia aspekty.

Kompilator::Debug::Pascal::Leksyka : Wypisuje przyjętą klasyfikację oraz rozpoznany zbiór leksemów

Kompilator::Debug::Api::DrzewoPlain : Wypisuje strukturę drzewa rozbioru syntaktycznego w czystej formie tekstowej

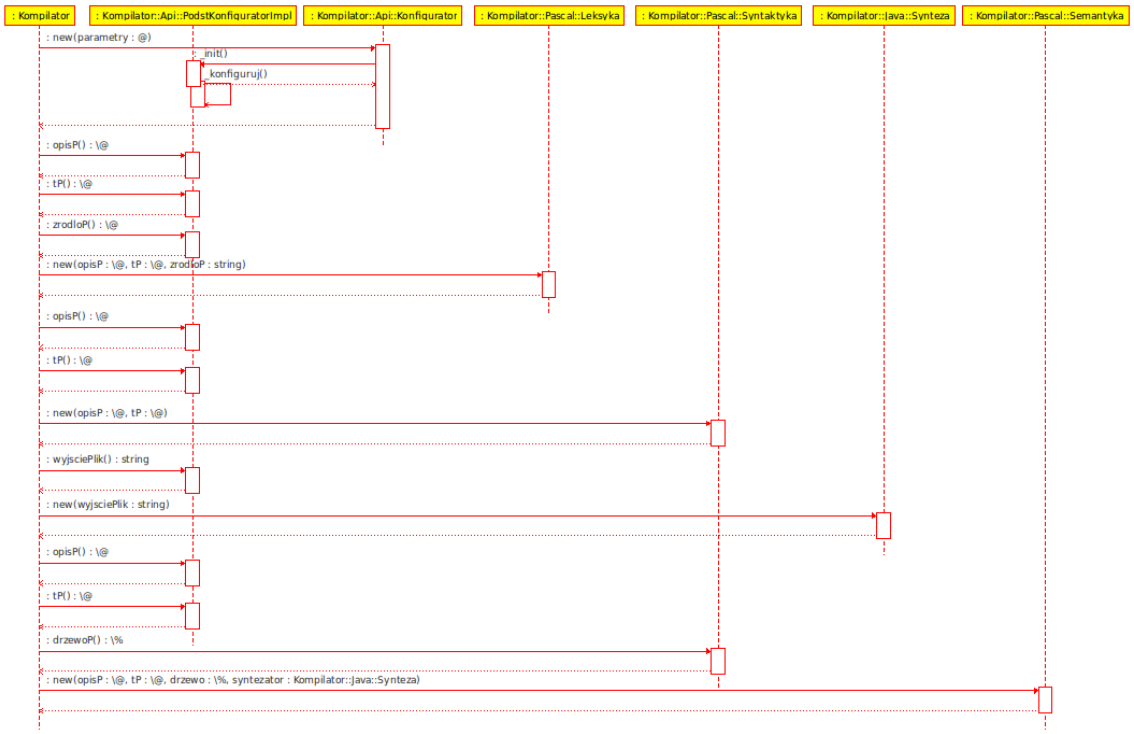
Kompilator::Debug::Api::DrzewoSvg : Buduje strukturę drzewa w pliku XML korzystając z szablonu http://weston.ruter.net/projects/syntax-tree-drawer/

Kompilator::Debug::Java::Synteza : Drukuje wywołania funkcji wraz z przyjętymi argumentami dla klas z nazwą Kompilator::Java::Synteza::

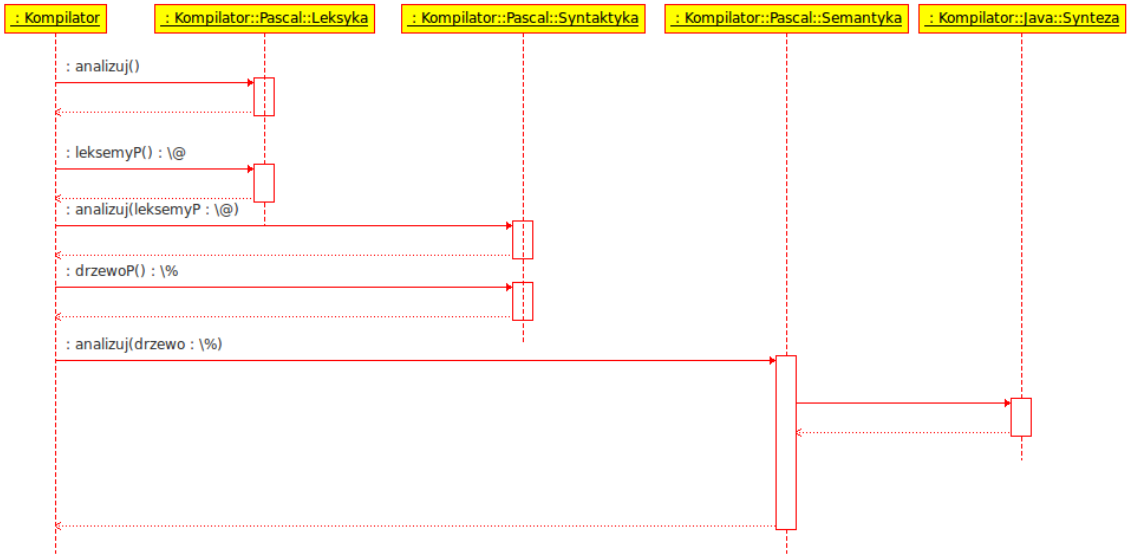
Kompilator::Debug::Java::Synteza : Drukuje wywołania funkcji wraz z przyjętymi argumentami dla klas z nazwą Kompilator::Java::Synteza::

Kompilator::Debug::Api::Synteza::ONP : Informuje o działaniach na stosie ONP

4.2.2 Sekwencje



Rysunek 9: Budowa świata



Rysunek 10: Kompilacja

4.3 Zastosowanie

Skompilujmy kod źródłowy

Listing 7: Przykładowy kod źródłowy

```
1  var zm1,zm2:integer;
2    var zm3:float;
3  var napis :
4  string;
5  proc liniowa(x:integer) : integer
6  begin
7    return 4 + 3.5 * x;
8  end
9  proc funkcja(zm1:float,zm2:integer) : integer
10 begin
11   var zm3:integer;
12   zm3 = 2 * zm1 + 3 * zm2;
13   zm1 = call liniowa( 5 );
14   return zm1 + zm2 + zm3;
15 end
16 start
17   zm1 = 3 * 4;
18   zm2 = call funkcja(call liniowa(2+zm1),4);
19 end
```

Wynik uzupełniony o komentarz

```
$ perl Kompilator.pl zrodloPlik=/zrodla/in.pas
SuccesFull
real 0m4.685s # Intel Atom N270
user 0m4.600s # Ubuntu 10.04 Lucid Lynx
sys 0m0.064s     # 2.6.32-28-generic
$ cat A.class

# cialo procedury liniowa
ldc 0 # laduje pierwsza stala 4, bedzie intem
ldc 1 # laduje druga stala 3.5, bedzie floatem
iload 0 # laduje przekazany parametr jako int
i2f # konwertuje parametr na float
fmul # mnozy szczytowe liczby zmiennoprzecinkowe
f2i # wynik konwertuje do inta
iadd # dodaje 2 szczytowe inty
# brakuje return
```

```
# cialo procedury funkcja
iload 2 # laduje wartosc z trzeciego parametru, zm3 int
ldc 2 # laduje trzecia stala 2, bedzie intem
fload 0 # laduje pierwszy parametr, zm1 float
f2i # konwertuje wartosc parametru na int
imul # mnozy inty ze szczytu 2 * zm1
ldc 3 # laduje czwarta stala 3, typ calkowity
iload 1 # laduje drugi parametr, zm2 int
imul # mnozy inty ze szczytu 3 * zm2
iadd # dodaje wyniki obu mnozen
istore 2 # zapisuje wynik pod trzecia zmienna zm3 jako int

fload 0 # laduje pierwszy parametr zm1 jako float
ldc 4 # laduje piata stala 5, bedzie intem
invoke 0 # wywoluje procedure liniowa
i2f # wynik wywolania konwertuje na float
fstore 0 # zapisuje go w zm1

fload 0 # laduje zm1 jako float
iload 1 # laduje zm2 jako int
iload 2 # laduje zm3 jako int
iadd # dodaje inty zm2 + zm3
i2f # wynik konwertuje na float
fadd # dodaje floaty zm1 + (zm2+zm3)
#brakuje return


# wnetrze glownej czesci programu
getfield 0 # laduje zmienna globalna zm1, int
# blad powinien zaladowc parametr lokalny 0, ktory powinien byc referencja this
ldc 3 # laduje czwarta stala calkowita 3
ldc 0 # laduje pierwsza stala calkowita 4
imul # mnozy stale
putfield 0 # wynik zapisuje w zmiennej globalnej zm1
getfield 1 # laduje zmienna globalna zm2, int
ldc 2 # laduje trzecia stala 2, int
getfield 0 # laduje pierwsza zmienna globalna zm1, int
```

```
iadd # dodaje 2+zm1
invoke 0 # wywoluje pierwsza funkcje zagniezdzona, liniowa
ldc 0 # laduje pierwsza stala 4, int
invoke 1 # wywoluje druga funkcje zewnetrzna, funkcja
```

Na pewno brakuje instrukcji powrotów z funkcji. Niewłaściwie traktowane jest przypisanie wartości do pól klasy. Kompilator powinien ściągnąc z tablicy lokalnej argument 0, który powinien być referencją na "ten" obiekt. Parametry procedur powinny zaczynać się od pierwszego indeksu. ONP działa poprawnie, zagnieżdżone wywołania funkcji również.

Do zaimplementowania pozostają przede wszystkim instrukcje skoku goto. Od miejsc rozgałęzień w kodzie źródłowym należy zliczać produkowane bajty strumienia pośredniego do ponownego zejścia alternatywnych dróg. Ilość bajtów zapisać po ifie w strumieniu jako offset. Do tego zadania należałoby wyodrębić odrębną klasę jako pomocniczą dla Kompilator::Api::Semantyka::SemantykaBaza.

Brakuje wydruku opkodów obsługi tablic. Atutem jest, iż klasa Kompilator::Pascal::Semantyka nie pozwala przydzielić pamięci do zmiennej zaalokowanej oraz zwolnić nieprzydzieloną.

Ułatwieniem byłaby obsługa nawiasowania w wyrażeniach, Kompilator::Api::Semantyka::Zmienne są na taką okoliczność przygotowane. Wystarczy uwzględnić je w zewnętrznym pliku opisu języka.

Repozytorium projektu:

`git://krystianwojtas.eu/kompilator`

5 bibliografia

http://home.agh.edu.pl/~marcino/Uczelnia/Files/JFA/zajecia03JFA.pdf

http://home.agh.edu.pl/~tekomp/pages/referaty/referaty-studenckie/5-1.htm

http://pl.wikipedia.org/wiki/Analiza_sk%C5%82adniowa

http://pl.wikipedia.org/wiki/Parser_LR

http://www.ibm.com/developerworks/ibm/library/it-haggar_bytecode/

http://en.wikipedia.org/wiki/Java_bytecode_instruction_listings

http://en.wikipedia.org/wiki/Class_%28file_format%29

http://java.sun.com/docs/books/jvms/second_edition/html/ClassFile.doc.html

http://www.tutorialspoint.com/perl/perl_oo_perl.htm

http://weston.ruter.net/projects/syntax-tree-drawer/

http://onjava.com/pub/a/onjava/2004/01/14/aop.html