Język PL/PL

PODRĘCZNIK UŻYTKOWNIKA

PODRĘCZNIK DO JĘZYKA PL/PL I JEGO KOMPILATORA ORAZ WNIOSKI Z PRAC PROWADZĄCYCH DO ICH POWSTANIA

Mateusz Strzałkowski
Michaela Klimek
Dominik Kikla

Kraków 2022

Spis treści

Wstęp

Co to język plpl?

Język PL/PL to imperatywny język programowania kompilowany do asemblera. PL/PL to język z polskimi słowami kluczowymi (oraz komunikatami o błędach) ze składnią podobna do C. Charakterystyczną cechą tego języka jest możliwość definicji wielu punktów wejścia sterowania do procedur, co razem z polską składnią czyni ten język wyjątkowym na współczesnych języków programowania.

Punkty wejściowe w procedurze – a co to takiego? – przykład składni

Punkty wejściowe polegają na definiowaniu „podfunkcji” w procedurze (choć ściślej należałoby stwierdzić, że w jednej procedurze gnieździ się wiele pełnoprawnych funkcji)

```
/*przykład 1*/
procedura -> całk
{
    całk zmienna1;
    zacznij fun1(całk zmienna2, znak zmienna4);
       jeśli(zmienna2 < 0) { zmienna2 = -zmienna2; }
    zacznij fun2(całk zmienna3)
   wypisz(zmienna4); wypisz(zmienna3);
    zwróć(zmienna2);
}
procedura
{
   zacznij program();
   fun1(2, 'a');
   zwróć();
}
```

fun1 i fun2 to są te „podfunkcje” w procedurze, czyli możemy zawołać fun1 a także fun2, bez deklarowania osobnej funkcji do tego, wystarczy dopisać punkt wejściowy z argumentami, może to być wykorzystane np. do walidacji danych (punkt wejściowy, który waliduje dane i pod nim punkt wejściowy, które te dane przetwarza i możemy wywołać od razu punkt wejściowy, które te dane przetwarza na poprawnych danych).

„zacznij program()” jest to odpowiednik do main() w wielu językach programowania, czyli „rozruchowy punkt programu” oznacza, że od tego miejsca program ma zacząć się wykonywać.

Kolejny przykład demonstruje tego mechanizmu w pełni:

```
/*Demonstarcja działania wielokrotnych punktów wejściowych do procedury*/
procedura{ zacznij program();
    g(1,2);
    h(1,2);
    f(1,2,1+2);
    zwróć();
}
procedura
{
    całk c,d;
    c=0;
    zacznij g(całk a, całk b);
```

```
    zacznij h(całk b, całk a);
    zacznij f(całk a, całk b, całk c);
    pisz("a:",a,", b:",b,", c:",c,"\n");
    zwróć();
}
```

Wyjście z tego programu:

```
$ ./wielo
a:1, b:2, c:0
a:2, b:1, c:0
a:1, b:2, c:3
```

Przykładowe programy napisane w języku plpl:

- Program wypisujący „Witaj Świecie” na wyjście

```
/*Witaj Świecie!*/
procedura
{
   zacznij program();
   wypisz(„Witaj Świecie!”);
}
```

- Program obliczający silnię na dwa sposoby:

```
/* Procedura mogąca obliczać silnię na dwa sposoby */
procedura -> całk{
    zacznij rek(całk n);
    jeśli(n<=1){zwróć(1);}
    zwróć(rek(n-1)*n);
    zacznij iter(całk n);
    całk wynik; wynik=1;
    dopóki(n > 0)
    {
        wynik = wynik * n;
        n = n - 1;
    }
    zwróć(wynik);
}
procedura{
    zacznij program();
    całk i,wr,wi; i=0;
    dopóki(i<10)
    {
        pisz("\n",i,":",rek(i)," : ",iter(i));
        i=i+1;
    }
    zwróć();
}
```

Instrukcja instalacji i użycia

Wymagania by użyć kompilatora języka plpl:

- **java JVM (w zmiennej środkowiskowej path)**
- **gcc 32 bitowy (w zmiennej środkowiskowej path)**
- **nasm 32 bit (w zmiennej środkowiskowej path)**

Jak używać kompilatora języka plpl?

1. Pobierz zipa z repozytorium https://gitlab.kis.agh.edu.pl/dpalka/komp22-kompilatorpl.git
2. Rozpakuj go do wybranego folderu. Zobaczysz plik plplk.
   Plik plplk jest bashowym proxy więc wystarczy dodać go do PATH (np.:export PATH=$PATH:/c/...reszta ścieżki .../PLPL)
3. Potem wywołać z dowolnego miejsca (w bashu):
   plplk -i plik_źródłowy.plpl*
   Ewentualnie można wołać bezpośrednio Javę:
   java -jar ..../kompilatorPLPL.jar ...argumenty...

Aplikacja kompilatora produkuje plik .asm - w asemblerze NASM. Potem próbuje wołać nasm z odpowiednimi argumentami i na koniec gcc w charakterze linkera(tekst wydanych przez kompilator komend pojawia się na jego standardowym wyjściu).
Istnieje możliwość ręcznego użycia innego linkera (np. ld), lecz wymagałoby to jawnego podania plików potrzebnych do ustanowienia środowiska wykonawczego C. (np. crt0)

* proponujemy rozszerzenie .plpl ()

Tryb interaktywny kompilatora

Opcja -I uruchamia debugger samego kompilatora, czyli po prostu linię komend pozwalającą interaktywnie zobaczyć interaktywnie, co rzeczony stwór ma w środku, żeby nie zastanawiać się, na podstawie wyników, co siedzi w ukrytych warstwach abstrakcji aplikacji kompilatora.

Co do zasady Debugger nie robi nic - ma nie zmieniać stanu właściwego kompilatora i tylko przyczepia się do wywołań listenerów.

Żeby skłonić do działania ów debugger, umieszczamy w odpowiednim miejscu w kodzie źródłowym znacznik w nawiasach ostrych, np. <S>, co znaczy, że podczas przebiegu analizy semantycznej, ma się zacząć sesja interaktywna w kontekście danego miejsca w kodzie (procedura, zakres).
Komendy, które można wpisać, przedstawiają się mniej więcej tak:
, - przecinek - zatrzymaj się przy następnym znaczniku
dokończ/**dok.** - ubija debuger i daje dokończyć kompilację
zgiń/` - ubija całą aplikację
cd proc 1 - przejdź do procedury 1
symbole - przestrzeń nazw
symbole tu - symbole tylko w aktualnym zakresie
symbol <nazwa> - szuka symbolu, można dodać -l
ls - dużo różnych rzeczy wypisuje
ls proc - listuje procedury
ramka - wypisuje ramkę aktualnej procedury

To jest wewnętrzne narzędzie dla twórców kompilatora, właściwie do sprawdzania wyników przebiegu analizy semantycznej (zbierania deklaracji) i jeśli trochę, albo bardzo nie chce działać, to trudno.

Opis składni

Pełny odpis prawdziwej składni, w notacji antlr4 przytoczony jest na końcu tego dokumentu.

Język PL/PL, jest językiem programowania – PL, imperatywnym i mocno typowanym ze składnią naśladującą język polski (stąd drugie PL), zbliżoną do C, albo raczej do utartych praktyk z niego wyrosłych.

Podstawowa składnia wewnątrz procedur:

Proponowana składnia jest istotnie zbliżona do C, wyjąwszy kwestie procedur/funkcji oraz wskaźników, zamiast których obecne są referencje i nie ma również unarnych operatorów * i &.

Sam kod wewnątrz procedury składa się z instrukcji, oddzielonych średnikami. Białe znaki są ignorowane, prócz tego, że rozdzielają tokeny. Instrukcja może być instrukcją prostą, instrukcją złożoną, jedną z instrukcji sterujących przepływem sterowania, bądź deklaracją.

Deklaracje obiektów

Wszelkie obiekty (prócz anonimowych, pojawiających się w toku obliczania wyrażeń), przed użyciem muszą być zadeklarowane w następujący sposób: <nazwa typu> <identyfikator>; W niektórych sytuacjach możliwa jest jeszcze inicjalizacja :
<nazwa typu> <identyfikator> = <dozwolona wartość typu>;

Identyfikator

W tym języku identyfikator może składać się ze znaków polskiego alfabetu, małych lub wielkich, podkreślnika i ewentualnie po choć jednym znaku alfabetu, z cyfr. Nazwa taka odnosi się do obiektu, o określonym podczas kompilacji typie.

Typy

W tym języku typy dzielą się zasadniczo na typy atomiczne (proste) i referencje.

Typy atomiczne

są podyktowane powszechną architekturą sprzętu i przyzwyczajeniami.
znak – to 8 bitów – char, z literałami typu ‘ł’, ‘a’
całk - 32 bity – int, ze standardowa arytmetyką

Referencje

do tablicy – zapewniony operator dostępu do elementu [] - zwraca wartość np.: t[2][a];
do obiektu – zapewniony selektor (kropka) np.: a.c

Tablice

Deklaracja *całk a[7];* tworzy tablicę i nazwa *a* oznacza referencję do niej. Do elementu tablicy dostęp zapewnia operator nawiasów kwadratowych. Nazwa *a* nie jest równoważna z adresem zerowego elementu (jak w C), nie ma wskaźników.

Typy złożone

Przykłady:
typ zespolona{rzeczyw cz_rzeczywista; rzeczyw cz_zespolona;};
typ kot{całk wiek=2; rzeczyw wredność=+inf; funkcja zamiaucz = kotomiauczenie};

pokazują, że struktury definiowane za pomocą słowa kluczowego „typ”, zachowują się bardzo podobnie do „typedef struct” w C.

Deklaracja: *zespolona b;* deklaruje w rzeczywistości referencję (pustą), niezainicjalizowaną. **nic** jest wyróżnionym literałem, odpowiadającym NULL. Instancjacja obiektu odbywa się konstrukcją w formie: *zespolona b = nowa(zespolona);* Konstrukcja **nowa(**<nazwa typu>**)**; powoduje utworzenie na stercie nowego obiektu. **zapomnij(**<referencja do obiektu>**)**; powoduje zwolnienie pamięci na stercie.

Struktury kontroli przepływu sterowania

Przykład języka C pokazuje, że do budowania kodu imperatywnego wystarczą w istocie dwie struktury – warunek i pętla, odpowiednio w C *if* i *while*. PL/PL zapewnia ten minimalny zbiór.
Zarys gramatyki prostego kodu imperatywnego:
<lista instrukcji> := <instrukcja>+
<instrukcja> := <instrukcja wyboru> | <instrukcja pętli> | <instrukcja złożona> | <instrukcja prosta> | <instrukcja wkroczenia> | <instrukcja powrotu> | <deklaracja atomiczna>
<instrukcja wyboru> := **jeśli(**<wyrażenie>**)**<instrukcja> **inaczej** <instrukcja>
<instrukcja pętli> := **dopóki(**<wyrażenie>**)**<instrukcja>
<instrukcja złożona> := **{**<lista instrukcji>**}**
<instrukcja prosta> := <wywołanie>**;** | <przypisanie>**;**
<przypisanie> := <identyfikator> **=** <wyrażenie>**;**
<deklaracja atomiczna> := <nazwa typu> <identyfikator>**;**
| <nazwa typu> <identyfikator> **=** <literał>**;**
Instrukcje wkroczenia i powrotu opisane są później.

Wyrażenia

Wyrażenia mają działać zgodnie z zasadami arytmetyki i z oczekiwaniami (językiem C).
Zatem *i = 2+7*(2-5*a);* powinien być poprawnym napisem, o ile *i* oraz *a* są typami atomicznymi.
Na referencjach nie ma zdefiniowanych operacji arytmetycznych, więc ich występowanie bez selektorów ([] dla tablic, albo kropka dla obiektów) jest ograniczone do wywołań funkcji z argumentami odpowiedniego typu i przypisania wartości referencji zwracanych przez funkcje.
f(a,b,c); a=f(a,b,c);

Operatory

arytmetyczne: *** / % + -** (bez % dla liczb zmiennoprzecinkowych),
logiczne: && || **!** (AND OR NOT), pracujące na liczbach całkowitych, gdzie 0 – fałsz, 1 – prawda.

<wyrażenie> := **(**<wyrażenie>**)** | <wyrażenie> <operator dwuargumentowy> <wyrażenie> | <identyfikator $_{\text{(obiektu atomicznego)}}$>

Procedury i funkcje

W tym miejscu opisywany język odchodzi od form obecnie powszechnych, nie posiadając składni definicji funkcji jako takiej. Zdecydowaliśmy się bowiem odkopać archaiczny pomysł, obecny niegdyś w PL/I, czy COBOLu – wielokrotne punkty wejścia (multiple entry points). W tych językach istniała mniej więcej *zwyczajna* składnia procedury, z nagłówkiem zawierającym nazwę (i informacje o parametrach), a po niej blok kodu, tyle, że w tym kodzie mogło wystąpić specjalne słowo kluczowe (ENTRY <nazwa>). Instrukcja CALL <nazwa> powodowała wskoczenie sterowania do funkcji, ale w miejscu zaznaczonym owym ENTRY, a nie na początku.

Można sobie oczywiście wyobrazić, że jeśli istnieje byt zwany funkcją, z jednym punktem wejścia i wieloma punktami wyjścia (wieloma instrukcjami powrotu, *return*), to możliwe jest opisanie bytu posiadającego wiele punktów wejściowych, czy też wkroczenia sterowania, bez wyróżniania jakiegokolwiek z nich. Taki model jest realizowany w języku PL/PL.

Najogólniejszym stopniem organizacji kodu jest zatem pojedynczy, ograniczony nawiasami (wąsatymi) blok kodu, z zaznaczeniem, jaki typ zwracany jest przy powrocie z niego – gdyby różne instrukcje powrotu zwracały różne typy, powstałby ciężko rozwiązywalny nieporządek.
Ogólna składnia zatem przyjmuje postać:
procedura (**->** <nazwa typu (zwracanego)>)? {<lista instrukcji>}
Nawias przed strzałką, po którym następuje znak zapytania jest tu elementem metajęzyka, żeby opisać, że kiedy procedura nic nie zwraca, strzałka i nazwa typu nie są potrzebne. Na przykład:
procedura -> całk {<lista instrukcji>}
procedura {<lista instrukcji>} - nic nie zwraca.

Celem zaznaczenia punktów wejścia i wyjścia, istnieją dwie wyróżnione instrukcje:
<instrukcja powrotu> := **zwróć(**<identyfikator (o typie zgodnym z sygnaturą na górze procedury)>**);**
<instrukcja wkroczenia> := **zacznij** <identyfikator – nazwa „funkcji”>**(**<lista parametrów formalnych>**);**

Instrukcja powrotu, działa zgodnie z obyczajem, umożliwiając powrót sterowania do procedury wywołującej, zabierając ze sobą wartość zwracaną.

Instrukcja wkroczenia jest jakoby deklaracją funkcji – **zacznij** f(**całk** a); sprawia, że w globalnej przestrzeni nazw pojawia się *referencja do wkroczenia*, pod nazwą f. Można zatem, z dowolnego miejsca wywołać f(7); Sterowanie wkroczy do funkcji, w miejscu oznaczonym przez odpowiednie **zacznij**, skopiuje wartości przekazanych argumentów do zmiennych lokalnych i będzie wykonywać kod procedury, do napotkania instrukcji **zwróć**, ignorując (przeskakując) wszystkie kolejno napotkane instrukcje wkroczenia.

Procedura z *n* punktami wejścia i pewną liczbą punktów wyjścia, jeśli się zastanowić, równoważna jest *n* zwyczajnym funkcjom, z fragmentami (lub większością) nakładającego się kodu. Z tego też powodu, *referencję do wkroczenia*, można istotnie nazwać *funkcją*.

Przykłady procedur:

```
procedura -> całk
{
   całk w;
   zacznij proporcjonalna(całk a, całk x);
   całk b=0;
   zacznij liniowa(całk a, całk b, całk x);
   w = a*x+b;
   jeśli(a<0)
   {
      pisz("Funkcja zwraca liczbe ujemną");
      zwróć(w);
   }
   inaczej{piszf("Funkcja zwraca %d", w); zwróć(w);}
}
```

Powyższa procedura ilustruje opisaną składnię, dając jeden z prostszych przykładów jej wykorzystania, do zapewnienia ekwiwalentu parametrów domyślnych. Postulujemy również, jeśli nie sprawi to technicznych problemów, możliwość przeładowania nazw funkcji/wkroczeń (identyfikację nie tylko po nazwie, ale też po typach parametrów), co uczyniłoby tę składnię może nawet użyteczną.

Chwila uwagi pozwala zauważyć problem wymagający doprecyzowania - czy linie:

całk b=0;
zacznij liniowa(**całk** a, **całk** b, **całk** x);

zawierają redeklarację zmiennej b?

Rozwiązaniem tego problemu byłby sztywny wymóg zgromadzenia deklaracji na początku procedury i usunięcie nazw typów z list parametrów:

całk a,b,x; **zacznij** proporcjonalna(a,x); b=0; **zacznij** liniowa(a,b,x);
co pozostaje opcją, w razie trudności implementacyjnych, zdecydowaliśmy się jednak przyjąć zasadę pozwalającą na pseudo – redeklarację, jeśli typ się zgadza i jest ona umieszczona na liście parametrów (a nie w ciągu instrukcji).
Zatem **całk a; zacznij** f(**całk** a); **zacznij** f(**całk** a, **całk** b); jest konstrukcją dozwoloną, ale
zacznij f(**całk** a); **całk a; zacznij** f(**całk** a, **całk** b); albo
zacznij f(**całk** a, **całk** b); **całk a; zacznij** f(**całk** a); już nie są dozwolone, ponieważ następuje w obu ostatnich przypadkach powtórna deklaracja, niebędąca częścią listy parametrów.

W powyższym przykładzie istnieje tylko jedna zmienna *a* (deklaracja na liście formalnej umieszcza identyfikator w przestrzeni nazw całej procedury, niezależnie od zagnieżdżenia), w odróżnieniu, od konstrukcji znanej z C:

```
int a=2;
{
        int a=3;
}
```

która, jak sądzimy, też powinna zostać zachowana w opisywanym języku.

Przykład programu:

```
procedura -> całk
{
    całk odp;
    zacznij ack(całk m, całk n);
    jeśli(m==0){odp = n+1;}
    inaczej jeśli(n==0){odp = ack(m-1, 1);}
    inaczej{odp = ack(m-1, ack(m,n-1));}
    zwróć(odp);
}

procedura
{
    całk i,j;
    zacznij program();
    i=0;
    dopóki(i<3)
    {
        j=0;
        dopóki(j<3)
        {
            pisz("Wartość funkcji Ackermana(", i,",",j,
                ") to",ack(i,j));/* to jest kontynuacja poprzedniej linii*/
            j=j+1;
        }
        i=i+1;
    }
    //niejawne zwróć na końcu procedury
}
```

Jak widać, **zacznij program**(); nadaje się na oznaczenie punktu wejścia do programu, a osobliwa składnia procedur/wielo-funkcji łatwo redukuje się do równoważnika zwykłych funkcji.

Wnioski wyciągnięte z przebiegu projektu

Oryginalne „Uwagi dotyczące implementacji” zawarte we wstępnym opisie języka.

Mamy na celu, wykorzystując ANTLR, napisać kompilator tego języka do pewnego asemblera (języka niskiego poziomu). Nie wykluczamy możliwości napisania własnej maszyny wirtualnej, wykonującej bezpośrednio ten asembler, byłaby użyteczna przy debugowaniu, chociaż nie uważamy tego za konieczne. Ze względu na charakter przedsięwzięcia, jego maksymalną

możliwą objętość i z definicji ograniczoną użyteczność, nie uznajemy za priorytetowe kwestii przenośności generowanego kodu/samego kompilatora, czy optymalizacji. Translacja do asemblera wyrażeń, instrukcji zawierających proste zmienne i opisanych powyżej podstawowych struktur sterowania przepływem, wydaje się być zadaniem wykonalnym w dostępnym, ograniczonym czasie natomiast implementacja szerszej gramatyki języka (np. referencji do funkcji), może zrodzić nieprzewidziane trudności i również nie jest niezbędna aby język stanowił funkcjonalną całość. Niewątpliwą zaletą byłoby zapewnienie choćby częściowej dostępności funkcji standardowej biblioteki języka C (GNU libc), co jest teoretycznie możliwe.

Niniejszy projekt jest w istocie studium możliwości stworzenia języka ogólnego przeznaczenia z różnym od powszechnego konceptem funkcji i procedur oraz wykorzystania substratu innego niż angielski do konstrukcji gramatyki. Studium to zasadniczo kończy się sukcesem, choć niepełnym i niepozbawionym ułomności.

W ciągu półrocza udało nam się opracować składnię języka, zastosować Antlr do jego analizy składniowej, zaprojektować architekturę aplikacji, opartej na dostarczonym przez Antlr froncie, przeprowadzającą analizę semantyczną oraz generację kodu w języku docelowym – Netwide Assembler na podzbiór architektury IA-32, oraz wytworzyć implementację takiego projektu. Powstał w ten sposób zrąb języka z działającym kompilatorem, co stanowi o ogólnym sukcesie przedsięwzięcia.

Jednakże z braku czasu opuściliśmy pewne elementy nie będące niezbędne do funkcjonowania języka:

- Liczby zmiennoprzecinkowe, ponieważ wymogłyby w praktyce podwojenie objętości schematów generacji kodu, jako że jednostka FPU(x87) posiada własne rejestry i osobne instrukcje.
- Większość przypadków deklaracji z przypisaniem, ze względu na dualność semantyczną tych konstruktów i w konsekwencji dodatkowe zasoby potrzebne do rozsądnej implementacji.
- Referencje do funkcji, jako ze były określone jako cele dodatkowy, oraz poważnie zwiększyłyby złożoność tablic kompilatora, przetwarzania deklaracji i samej gramatyki. Ponadto, w przyjętym stylu składniowym, typy takie wcale nie byłyby czytelniejsze od wskaźników do funkcji w C.
- Drugi proponowany sposób wywołania, jako że większość zapotrzebowania nań generowałyby referencje do funkcji oraz symbole importowane z innych plików, oba koncepty niezrealizowane.
- Mechanizm typu import/include, ze względu na wielką złożoność implementacyjną oraz możliwości osiągnięcia najprostszych jego efektów bezpośrednio przy pomocy linkera.
- Upraszczanie wyrażeń podczas kompilacji.
- Poważny mechanizm rzutowania.

Udało się natomiast, przede wszystkim, pokonać główne wyzwanie implementacyjne wynikłe z gramatyki – wywołania procedur z wieloma punktami wejścia sterowania – zadanie o tyle trudniejsze, że o ile dla większości innych konstruktów obecnych w języku istnieją powszechnie dostępne schematy generacji kodu, lub są łatwo wywodliwe, o tyle ten przypadek wymagał napisania idiosynkratycznego schematu kodu.

Reszta planowanych prac, obejmujących proste zmienne, struktury sterowania, cały system deklaracji i zasięgów, zostały konsekwentnie wykonane. Ponadto zaimplementowaliśmy elementy oznaczone w oryginalnym projekcie jako opcjonalne – tablice (też wielokrotnie zagnieżdżone) oraz struktury (obiekty typów złożonych), oraz ich dynamiczną alokację.

W wyniku tych wysiłków nabyliśmy cenna wiedzę z zakresu planowania prac projektowych, i projektowania obszernych aplikacji, nauczyliśmy się pisać gramatyki, używać antlr, do pewnego

stopnia asemblera NASM uzyskaliśmy pewna wiedzę z zakresu linkowania i formatów binarnych. Spośród tego, wybija się kilka wniosków co do konstrukcji kompilatora:

1. Rozwinięte mechanizmy opisu semantyki i sprawdzania jej poprawności na danym drzewie są konieczne do utrzymywalności i czytelności generatora kodu. Jednocześnie składnia akcji wewnątrzgramatycznych, wartości zwracanych antlr, proste mechanizmy oznaczania węzłów wydają się niewystarczające do osiągnięcia tego zadania przy dużej objętości i stopniu skomplikowania informacji przechowywanych podczas kompilacji.
2. Jak język źródłowy reprezentowany jest przez drzewo składniowe, tak na wyjściu generatora powinna znajdować się pewna abstrakcja języka docelowego. Nawet wzorzec visitora z antlr, w połączeniu z bezpośrednio wplecionymi weń schematami kodu asemblerowego, staje się trudny w utrzymaniu. Ponadto reprezentacja abstrakcyjna pozwoliłaby na wiele języków docelowych, realizacje takich zadań, jak przydział rejestrów i budowę prostych mechanizmów optymalizacyjnych.

Sądzę, że projekt ten pokazuje, że dzięki narzędziom pokroju antlr, istotnie możliwe byłoby stworzenie pełnoprawnego kompilatora nowego języka programowania, w kilka osób przez parę roboczomiesięcy, stosując podobną metodologię, lecz konsekwentnie więcej abstrakcji architektonicznych, co zapewniłoby wymagany poziom niezawodności i pole do rozszerzalności języka (w przypadku PL/PL o referencje do funkcji, interfejsy, inteligentne tablice, definiowalne operatory, wbudowane wyrażenia regularne etc etc…). Uzyskany przez nas kod wolny od błędów i zanieczyszczeń nie jest, jednak przedstawia pewną użyteczność.

Pierwotne pytanie projektu: jak czytałoby się program napisany „po polsku”, jakie możliwości przedstawiałyby *multiple entry points* wyjęte z grobowców PL/I i COBOLa, pozostaje nierozwiązane, ponieważ nie zdążono niestety w proponowanym języku napisać dłuższego programu, co z pewnością pomogłoby poprawić ergonomię komunikatów o błędach i na przykład kwestię duplikatów słów kluczowych, politykę reagowania na brak polskich ogonków w tychże słowach i wiele innych kwestii decydujących o ergonomii, jak dotąd przecież nie wypracowanych dla języka polskiego.

Wszystkim testerom chcielibyśmy podziękować za cierpliwość

Bibliografia

Dr Terence Parr *The Definitive ANTLR 4 Reference*

N. Wirth *Algorytmy + struktury danych = programy (rozdz 5)*

William M. Waite, Gerhard Goods *Konstrukcja kompilatorów, Wydawnictwa Naukowo-Techniczne, Warszawa 1989*

Pełny odpis z gramatyki

```
lexer grammar Lekserpl; // note "lexer grammar"
ZNACZNIK_DEBUGGERA :   '<<' ~( '<') *? '>>' -> channel(333);
NIC : 'nic' | 'NIC' | 'Nic';
STATYCZN: 'statyczn'[yea];
AUTOMATYCZN : 'automatyczn'[yea];
STAL : 'sta'[\u0142][yea];
TCALK: 'ca'[\u0142]'k'('owit'([yea])?)?;
TRZECZYW: 'rzeczyw';
TZNAK: 'znak';
TREF: 'ref';
NOWY: 'now'[yea];
WYPISZ: ('wy')? 'pisz';
PRZERWIJ : 'przerwij';
KONTYNUUJ:'kontynuuj'|'nazad';
ZMIENN :  CALK'.'CALK; //zmiennoprzecinkowa liczba
CALK :   [0-9]+ ;   //zwykła liczba
ZNAK_DOSL
    :   '\'' ( EscapeSequence | ~('\''|'\\') ) '\''
    ;
NAPIS_DOSL
    :  '"' ( EscapeSequence | ~('\\'|'"') )* '"'
    ;
LINIA_ASEMBLERA : '$$' .*? '\r'? '\n';
fragment
EscapeSequence
    :   '\\' ('b'|'t'|'n'|'f'|'r'|'\''|'\\');
ID  :   ([A-Za-z_]|OGONKI)([0-9A-Za-z_]|OGONKI)* ;      // identyfikatory
fragment OGONKI : [\u0104\u0105\u0106\u0107\u0118\u0119\u0141-\u0144\u015A\u015B\u0179-\u017C\u00D3\u00F3];//
ąćęłńśźżóĄĆĘŁŃŚŹŻÓ - polskie ogonki
EOS: ';';//END-OF-STATEMENT - koniec instrukcji, po polsku ewentualnie Egzaltowany-Okropnie-Srednik
//więte z książki
LINE_COMMENT : '//' .*? (EOF|'\r'? '\n') -> skip ; // Match "//" stuff '\n'
COMMENT : '/*' .*? '*/' -> skip ; // Match "/*" stuff "*/
WS  :   [ \t\r\n]+ -> skip ; // ignorowanie białych znaków
```

```
grammar plpl;
import Lekserpl;
program : (byt_globalny)* EOF;//a co z EOF?
byt_globalny: procedura | deklaracja_typu | deklaracja_prosta;
procedura   :  'procedura' ('->' typ_zwracany)? '{' lista_instrukcji  '}';
typ_zwracany: pelny_typ;
deklaracja_typu   : 'typ'  ID  '{' ( deklaracja_prosta )+ '}';//użytkownik wprowadza nowy typ
deklaracja_prosta   :  deklaracja_atomiczna  |  deklaracja_referencji;//użytkownik deklarje obiekt istniejącego już typu
(atomicznego, bądź zdefiniowanego)
//deklaracja_atomiczna   : przydomki nazwa_typu_atom   ID (',' ID)*  EOS
//                |   przydomki nazwa_typu_atom   ID  '='  (CALK | ZMIENN | ZNAK_DOSL) (',' ID  '='  (CALK | ZMIENN |
ZNAK_DOSL))* EOS ;
deklaracja_atomiczna
               :   przydomki nazwa_typu_atom   (deklarator_atomiczny_z_przypisaniem | deklarator_bez_przypisania) (','
(deklarator_atomiczny_z_przypisaniem | deklarator_bez_przypisania))* EOS ;
deklarator_bez_przypisania : ID;
deklarator_atomiczny_z_przypisaniem : ID  '='  (CALK | ZMIENN {notifyErrorListeners("Liczby zmienoprzecinkowe jeszcze nie
zaimplementowane");} | ZNAK_DOSL);
deklaracja_referencji   :
     przydomki pelny_typ (deklarator_bez_przypisania|deklarator_zlozony_z_przypisaniem) (',' (deklarator_bez_przypisania|
deklarator_zlozony_z_przypisaniem))*  EOS ;
deklarator_zlozony_z_przypisaniem : ID  '='  (lwartosc | stala_tablicowa /*|TABLICA_CALK_DOSL*/);
lista_instrukcji   : instrukcja+;
instrukcja   :   instrukcja_wyboru
             |   instrukcja_petli
             |   instrukcja_zlozona
             |   instrukcja_przerwania_petli
             |   instrukcja_kontynuacji_petli
             |   wypisanie
             |   instrukcja_prosta
             |   instrukcja_wkroczenia
             |   instrukcja_powrotu
             |   instrukcja_zakonczenia
            //|   deklaracja_atomiczna
            //|   deklaracja_referencji
             |   deklaracja_prosta
             |   wstawka_asemblerowa;
instrukcja_zlozona  : '{'  lista_instrukcji?  '}';
instrukcja_wyboru   : ('jeśli'|'jesli'|'gdy') '(' wyrazenie ')' instrukcja  ('inaczej'  instrukcja)?;
instrukcja_petli   : 'dopóki' '(' wyrazenie ')' instrukcja;
instrukcja_powrotu   : 'zwróć' '(' wyrazenie? ')' EOS;
instrukcja_wkroczenia   : 'zacznij'  ID '(' lista_parametrow_formalnych ')' EOS;
```

```
instrukcja_zakonczenia : 'skończ' '(' wyrazenie ')' EOS;
instrukcja_przerwania_petli   : PRZERWIJ EOS;
instrukcja_kontynuacji_petli   : KONTYNUUJ EOS;
wypisanie : WYPISZ '(' wyrazenie? (',' wyrazenie)*  ')'EOS;//wbudowane wypisywanie
instrukcja_prosta  :   wyrazenie EOS;
wstawka_asemblerowa : LINIA_ASEMBLERA;//przede wszystkim dla celów testowych, realnie wklejanie bezpośrednio kodu będzie
mało przydatne.
lista_parametrow_formalnych : (deklaracja_parametru  (',' deklaracja_parametru)*)?;
deklaracja_parametru   :  przydomki ( nazwa_typu_atom | pelny_typ)   deklarator_bez_przypisania;
/* przykład z książki
expression
    :   expression mult='*' expression          # expressionMultExpression
    |   expression add=( '+' | '-' ) expression # expressionAddExpression
    |   VARIABLE '=' expression                 # expressionAssign
    |   '(' expression ')'                      # parenthesisedExpression
    |   atom                                    # atomExpression
    ;
*/
wyrazenie
          : wywolanie_funkcji                          #wyrazenieWywolanie
          | naiwne_wywolanie                          #wyrazenieWywolanieNaiwne
          | alokacja                                   #wyrazenieAlokacja
          | dealokacja                                 #wyrazenieDealokacja
          | lwartosc                                   #wyrazenieLwartosc//
          | wyrazenie selektor_tablicowy               #wyrazenieSelekcjaTablicowa
          | wyrazenie selektor_typu_zlozonego          #wyrazenieSelekcjiSkladowej
          | adr='@' wyrazenie                           #wyrazenieAdres //funkcje rodem z C będą i tak potrzebować
adresów....
          | neg='!' wyrazenie                          #wyrazenieNegacja
          | znak=('-'| '+') wyrazenie                  #wyrazenieZnak
          | <assoc=right> wyrazenie '^' wyrazenie {notifyErrorListeners("Operator potegowania jeszcze nie
zaimplementowany...");}       #wyrazeniePoteg
          | wyrazenie mult=('*' | '/' |'%') wyrazenie   #wyrazenieMult
          | wyrazenie addyt=('+' | '-') wyrazenie       #wyrazenieAddyt
          | wyrazenie logicz=('&&' | '||')wyrazenie    #wyrazenieLogicz
          | wyrazenie porownanie=('==' | '!=' | '>' | '<' | '<=' | '>=')wyrazenie    #wyrazeniePorownanie
          | <assoc=right> wyrazenie '=' wyrazenie                    #wyrazeniePrzypisanieZwykle
          | <assoc=right> wyrazenie '^=' wyrazenie {notifyErrorListeners("Operator potegowania jeszcze nie
zaimplementowany...");}                     #wyrazeniePrzypisaniePoteg
          | <assoc=right> wyrazenie mult=('*=' | '/=' | '%='){notifyErrorListeners("'*='  '/='  '%=' jeszcze nie
zaimplementowane...");} wyrazenie#wyrazeniePrzypisanieMult
```

```
          | <assoc=right> wyrazenie addyt=('+=' | '-=') {notifyErrorListeners("'+=' | '-=' jeszcze nie
zaimplementowane...");}wyrazenie      #wyrazeniePrzypisanieAddyt
          | stala_atomiczna                                  #wyrazenieStala
          //| ID                                               #wyrazenieId//wystarczy sama lwartość, której ID jest
szczególnym przypadkiem
          //NAPIS_DOSL //??
          | '(' wyrazenie ')'                                #wyrazenieNawiasy
          ;
alokacja: NOWY '('pelny_typ_dynamiczny')';//bez nawiasów robi sie niejednoznaczność
dealokacja:'zapomnij' '('wyrazenie')';
// lwartosc: ID (selektor_tablicowy)?(selektor_typu_zlozonego(selektor_tablicowy)*)*;//konstrukcje typu a.b[d+2][7][12+w]
 lwartosc: ID | stala_tablicowa | NIC;//  tablica_calk_dosl  | DOSLOWNA_LOSOWOSC;
 //do wyrażeń:
 //|wyrazenie selektor_tablicowy #wyrazenieSelekcjaTablicowa
 //|wyrazenie selektor_typu_zlozonego #wyrazenieSelekcjiSkladowej
 selektor_tablicowy   :  '['  wyrazenie  ']';
 selektor_typu_zlozonego   :  '.' ID ;

 wywolanie_funkcji   :  ID '(' lista_parametrow_aktualnych ')';
 naiwne_wywolanie : 'C.' ID '(' lista_parametrow_aktualnych ')';
 lista_parametrow_aktualnych : (wyrazenie  (',' wyrazenie)*)?;
 stala_atomiczna   :  CALK  |  ZMIENN {notifyErrorListeners("Liczby zmienoprzecinkowe jeszcze nie zaimplementowane");} |
ZNAK_DOSL ;
 stala_tablicowa   : NAPIS_DOSL;
//pelny_typ : ((nazwa_typu_atom ('[]')* ('[' CALK ']')+ ) | ID (('[]')* ('[' CALK ']')*) ) ;
pelny_typ : (nazwa_typu_atom | ID ) (nieokreslony_deklarator_tablicowy)* (okreslony_deklarator_tablicowy)?;
pelny_typ_dynamiczny : (nazwa_dynamicznie_alokowalnego_typu_atom | ID ) (obliczany_deklarator_tablicowy)?
(nieokreslony_deklarator_tablicowy)*;//do alokacji pamięci
nieokreslony_deklarator_tablicowy: '[']';
okreslony_deklarator_tablicowy: '[' CALK ']';
obliczany_deklarator_tablicowy: '[' wyrazenie ']';
przydomki : ((STATYCZN|AUTOMATYCZN)? STAL?) | (STAL? (STATYCZN|AUTOMATYCZN)?);
//nazwa_typu_atom   : 'całk' | 'rzeczyw' | 'znak';
nazwa_typu_atom : TCALK | TRZECZYW {notifyErrorListeners("Liczby zmienoprzecinkowe jeszcze nie zaimplementowane");} |
TZNAK | TREF;
nazwa_dynamicznie_alokowalnego_typu_atom : TCALK | TRZECZYW {notifyErrorListeners("Liczby zmienoprzecinkowe jeszcze nie
zaimplementowane");} | TZNAK | TREF;//po prawdzie to skrót, żeby nie zmieniać żeby oddzielić symbole gramatyczne w
deklaracjach i w alokacji
```