Metody programowania 2017

Zadanie dodatkowe na pracownię nr 4,5 i 6

Na listach zadań z Haskella implementowaliśmy prosty język programowania. Aby móc uruchamiać programy w tym języku napisaliśmy interpreter, czyli program, który traktuje inny program jako dane i stara się symulować jego zachowanie. Celem tej listy zadań będzie napisanie kompilatora, czyli programu, który też traktuje inny program jako dane, ale tłumaczy go na kod rozumiany bezpośrednio przez maszynę.

Taki kod maszynowy może wyglądać różnie dla różnych maszyn, więc musimy ustalić na czym będziemy chcieli uruchamiać nasze programy. Procesory w komputerach, których używamy na co dzień reprezentują architekturę x86 albo ARM. Są to architektury rejestrowe, gdzie procesor dysponuje dużą ilością (kilkanaście) rejestrów ogólnego przeznaczenia, czyli specjalnych komórek pamięci bezpośrednio w samym procesorze, na których wykonywane są wszystkie obliczenia. Kompilator dla takiej architektury musi podejmować decyzje które zmienne będą trzymane w rejestrach, a które w pamięci głównej, co komplikuje jego konstrukcję. Tego problemu nie ma dla architektur stosowych, gdzie dane trzymamy na stosie, a obliczenia możemy wykonywać na danych, które znajdują się na wierzchołku stosu. W tym zadaniu będziemy kompilować na jedną z takich maszyn, czyli procesor Motorola 6809.

Motorola 6809

Procesor Motorola 6809 został wprowadzony w 1978 roku i jest jednym z najbardziej zaawansowanych procesorów ośmiobitowych. Był wykorzystywany między innymi w komputerach TRS-80 Color Computer, czyli tak zwanych CoCo. Na potrzeby tego zadania, przyjmijmy że ma on cztery 16-bitowe rejestry: akumulator (D), wskaźnik instrukcji (PC), wskaźnik stosu użytkownika (U) oraz wskaźnik stosu systemowego (S)[1]. Procesor jest połączony z pamięcią operacyjną 16-bitową szyną adresową oraz 8-bitową szyną danych, czyli traktuje pamięć jako tablicę składającą się z 65536 bajtów. Podczas pracy, procesor w kółko wykonuje następujące kroki:

1. pobiera instrukcję znajdującą się w pamięci pod adresem zapisanym w rejestrze PC, jednocześnie zwiększając wartość w rejestrze PC, tak by wskazywała na następną instrukcję;
2. pobiera dane potrzebne do wykonania pobranej instrukcji (może to wymagać kolejnego dostępu do pamięci);
3. procesor wykonuje instrukcje, np. dodaje dwie liczby;
4. procesor zapisuje wyniki w odpowiednich rejestrach lub pamięci.

Niektóre instrukcje mogą modyfikować wartość rejestru PC. Oznacza to, że następną instrukcją wykonaną przez procesor nie musi być instrukcja która w pamięci znajduje się jako kolejna. Mówimy wtedy, że procesor wykonuje skok.

Instrukcje procesora są kodowane jako odpowiednie sekwencje bajtów (na jedną instrukcję składa się od jednego do pięciu bajtów). Np. program, który dodaje dwie liczby ze stosu użytkownika i umieszcza wynik na tym stosie kodowany jest następującą sekwencją bajtów (zapisanych szesnastkowo):

```
EC C1 E3 C1 ED C3
```

Niestety taki zapis jest dla nas nieczytelny. Aby czytelnie zapisywać programy na maszynę posługujemy się specjalnym językiem, zwanym językiem asemblerowym albo po prostu asemblerem[2]

[1] Procesor 6809 ma jeszcze dwa 16-bitowe rejestry indeksowe X oraz Y, dwa 8-bitowe specjalne rejestry CC oraz DP, a akumulator D dzieli się na dwie 8-bitowe połówki A oraz B.

[2] Słowo *asembler* oznacza też program, który tłumaczy język asemblerowy na kod maszynowy.

w którym każda instrukcja przekłada się na jedną instrukcję kodu maszynowego. Powyższy program można zapisać w asemblerze następująco:

```
LDD   ,U++
ADDD  ,U++
STD   ,--U
```

Niektóre instrukcje (każda w naszym przykładzie) mają jeden lub więcej parametrów które oznaczają skąd wziąć dane do wykonania instrukcji. Np. `ADDD ,U++` oznacza: dodaj do rejestru D wartość znajdującą się w pamięci pod adresem, który jest w rejestrze U, jednocześnie zwiększając wartość rejestru U o 2. Procesor 6809 ma bardzo rozbudowany zestaw trybów adresowania, czyli tego jak mogą wyglądać parametry instrukcji wykonujących dostęp do pamięci. Aby nie opisywać wszystkich tych możliwości, tylko skupić się na tym co najistotniejsze dla kompilacji, wprowadziliśmy język pośredni, który na potrzeby tego zadania nazwiemy makroasemeblerem[3]. W tym języku nie będzie skomplikowanych trybów adresowania, a większość instrukcji będzie tłumaczyć się na jedną instrukcję procesora 6809. Nasz przykładowy program w makroasemblerze wygląda następująco:

```
MPopAcc
MAdd
MPush
```

Makroasembler

Poniżej opiszemy zestaw makroinstrukcji makroasemblera, do którego będziemy kompilować programy. Skompilowane programy będziemy reprezentować w Haskellu jako listy makroinstrukcji, natomiast pojedynczą makroinstrukcję jako element algebraicznego typu danych `MInstr`.

Instrukcje arytmetyczne i logiczne

Instrukcje arytmetyczne i logiczne pozwalają wykonywać proste obliczenia. Są one zawsze bezparametrowe, a dane do obliczeń biorą zawsze z akumulatora (rejestr D) i (dla operacji dwuargumentowych) ze stosu. Wynik umieszczany jest w akumulatorze. Dla operacji dwuargumentowych, pierwszy argument jest zawsze zdejmowany ze stosu (brany z pamięci pod adresem zapisanym w rejestrze U, zaś sam rejestr U zwiększany tak, by wskazywał następną wartość na stosie), a drugi z akumulatora. Taka kolejność pozwala na łatwe zaimplementowanie ewaluacji od lewej do prawej.

Poniższa tabela przedstawia wszystkie instrukcje arytmetyczno-logiczne.

Instrukcja	Pseudokod
`MNot`	`D := not D`
`MNeg`	`D := - D`
`MAnd`	`D := Memory[U,U+1] and D; U := U + 2`
`MOr`	`D := Memory[U,U+1] or  D; U := U + 2`
`MAdd`	`D := Memory[U,U+1]  +  D; U := U + 2`
`MSub`	`D := Memory[U,U+1]  -  D; U := U + 2`
`MMul`	`D := Memory[U,U+1]  *  D; U := U + 2`
`MDiv`	`D := Memory[U,U+1] div D; U := U + 2`
`MMod`	`D := Memory[U,U+1] mod D; U := U + 2`

Uwaga: instrukcje `MNot`, `MAnd` oraz `MNot` są instrukcjami bitowymi: obliczają one odpowiednią operację logiczną równolegle na odpowiadających sobie bitach swoich argumentów.

Wszystkie te operacje działają na słowach 16-bitowych, ale sama pamięć jest adresowana bajtowo. Zatem każda z tych operacji (za wyjątkiem `MNot` i `MNeg`) wymaga odczytania dwóch bajtów: starszych ośmiu bitów argumentu (pod adresem U) i młodszych ośmiu bitów (pod adresem U+1) słowa 16-bitowego.

[3] Słowo *makroasembler* oznacza zwykle język asemblerowy, w którym można definiować *makroinstrukcje*, czyli bardziej złożone instrukcje, które tłumaczą się na kilka instrukcji zwykłego asemblera. W tym zadaniu nasz makroasembler ma wbudowane pewne makroinstrukcje, ale nie pozwala na definiowanie nowych.

Manipulowanie stosem

Mamy pięć instrukcji do manipulowania stosem użytkownika.

Instrukcja	Pseudokod i opis
`MGetLocal` n	`D := Memory[U+2`n`,U+2`n`+1]` Pobiera n-te słowo ze stosu do akumulatora
`MSetLocal` n	`Memory[U+2`n`,U+2`n`+1] := D` Zapisuje akumulator jako n-te słowo na stosie
`MPush`	`U := U - 2; Memory[U,U+1] := D` Odkłada akumulator na stos
`MPopN` n	`U := U + 2`n Usuwa n słów ze stosu
`MPopAcc`	`D := Memory[U,U+1]; U := U + 2` Zdejmuje wartość ze stosu i zapisuje ją w akumulatorze

Etykiety i skoki

Wśród konstruktorów typu `MInstr` znajduje się konstruktor `MLabel` przyjmujący parametr typu `Label` (zdefiniowanym jako `Int`) zwany etykietą. Etykieta l nie generuje żadnej instrukcji, ale jest informacją dla asemblera, że l oznacza adres następnej instrukcji.

Etykiety są przydatne dla instrukcji skoku, ponieważ można w czytelny sposób zadać miejsce w kodzie do którego procesor powinien skoczyć (podając etykietę zamiast adresu). W omawianym makroasemblerze mamy trzy instrukcje skoku.

Instrukcja	Pseudokod i opis
`MBranch` cc l	`if` cc `then PC :=` l Skocz do etykiety l jeśli spełniony jest warunek cc
`MJump` l	`PC :=` l Skocz do etykiety l (bezwarunkowo)
`MJumpAcc`	`PC := D` Skocz do adresu zapisanego w akumulatorze

Dla instrukcji skoku warunkowego (`MBranch`) można podawać tylko proste warunki, takie jak sprawdzenie znaku wartości w akumulatorze albo porównanie wartości ze stosu z wartością w akumulatorze. Poniższa tabela opisuje wszystkie możliwe warunki. Sprawdzanie dowolnego z ostatnich sześciu warunków modyfikuje wskaźnik stosu użytkownika U.

Warunek	Efekty uboczne (pseudokod)	Kiedy spełniony?
`MC_1`		zawsze
`MC_0`		nigdy
`MC_Z`		jeśli `D` $=$ `0`
`MC_NZ`		jeśli `D` $\neq$ `0`
`MC_P`		jeśli `D` $>$ `0`
`MC_NP`		jeśli `D` $\leq$ `0`
`MC_N`		jeśli `D` $<$ `0`
`MC_NN`		jeśli `D` $\geq$ `0`
`MC_EQ`	`tmp := Memory[U,U+1]; U := U + 2`	jeśli `tmp` $=$ `D`
`MC_NE`	`tmp := Memory[U,U+1]; U := U + 2`	jeśli `tmp` $\neq$ `D`
`MC_LT`	`tmp := Memory[U,U+1]; U := U + 2`	jeśli `tmp` $<$ `D`
`MC_GE`	`tmp := Memory[U,U+1]; U := U + 2`	jeśli `tmp` $\geq$ `D`
`MC_LE`	`tmp := Memory[U,U+1]; U := U + 2`	jeśli `tmp` $\leq$ `D`
`MC_GT`	`tmp := Memory[U,U+1]; U := U + 2`	jeśli `tmp` $>$ `D`

Instrukcja skoku warunkowego jest przydatna wyrażenia pętli oraz instrukcji warunkowych. Np. poniższy kod implementuje algorytm Euklidesa. Dane wejściowe znajdują się na stosie, a wynik w rejestrze D. W tym przykładzie `dont_swap` oraz `loop` są etykietami.

```
    MGetLocal  0
    MPush
    MGetLocal  2
    MBranch    MC_GE dont_swap
    MGetLocal  0
loop
    MPush
    MGetLocal  2
    MSetLocal  1
    MPopAcc
    MSetLocal  1
dont_swap
    MSub
    MPush
    MBranch    MC_NZ loop
    MPopN      1
    MPopAcc
```

Instrukcja `MJumpAcc` powinna być potrzebna tylko w najtrudniejszym wariancie zadania.

Procedury

Do zaimplementowania procedur i funkcji mamy specjalne instrukcje skoku które zapamiętują adres następnej instrukcji na stosie systemowym. Dzięki temu program może powrócić do miejsca z którego wykonał skok (wywołał procedurę). Instrukcje związane z implementacją procedur przedstawione są w poniższej tabeli:

Instrukcja	Pseudokod i opis
`MCall` l	`S := S - 2; Memory[S,S+1] := PC; PC :=` l Zawołaj procedurę znajdującą się pod etykietą l
`MCallAcc`	`S := S - 2; Memory[S,S+1] := PC; PC := D` Zawołaj procedurę, której adres znajduje się w akumulatorze
`MRet`	`PC := Memory[S,S+1]; S := S + 2` Powrót z procedury

Funkcje jednoargumentowe (tak jak mamy w tym zadaniu) można zaimplementować za pomocą tych instrukcji, przekazując argument i wartość przez akumulator. Jako przykład podajemy program który oblicza sumę kwadratów liczb znajdujących się na stosie:

```
    MPopAcc
    MCall     sqr
    MPush
    MGetLocal 1
    MCall     sqr
    MAdd
    MPopN     2
    MRet

sqr
    MPush
    MMul
    MRet
```

Stałe

Dowolna stała, lub adres etykiety może zostać załadowany do akumulatora, za pomocą następujących instrukcji

Instrukcja	Pseudokod i opis
`MConst` n	`D :=` n Załaduj stałą n do akumulatora
`MGetLabel` l	`D :=` l Załaduj adres etykiety l do akumulatora

Operacje na rekordach

Ostatnią grupą operacji, jaką mamy do dyspozycji, są operacje na rekordach. Powinny być one przydatne do zaimplementowania par, list oraz domknięć.

`MAlloc` n	`D := alloc(`n`)` Przydziel pamięć dla rekordu składającego się z n słów 16-bitowych
`MGet` n	`X := D; D := Memory[X+2`n`,X+2`n`+1]` Weź n-te pole rekordu znajdującego się w akumulatorze
`MSet` n	`X := Memory[U,U+1]; Memory[X+2`n`,X+2`n`+1] := D` Zapisz akumulator jako n-pole rekordu znajdującego się na stosie

Warto zwrócić uwagę na to, że instrukcja `MSet` nie zdejmuje adresu rekordu ze stosu. Takie zachowanie jest celowe, bo pozwala na łatwe inicjowanie kilku pól w rekordzie. Np. program tworzący parę `(13, 42)` wygląda następująco.

```
MAlloc  2
MPush
MConst  13
MSet    0
MConst  42
MSet    1
MPopAcc
```

Instrukcja `MAlloc` przydziela $2n$ bajtów na stercie, których adres umieszczany jest w rejestrze D. Można założyć, że przydzielony adres jest parzysty i różny od zera. Choć wydaje się ona skomplikowana, to jest tłumaczona na dokładnie dwie instrukcje procesora 6809! Dokładniej, tłumaczenie instrukcji `MAlloc` n wygląda następująco.

```
LDD   #(256*n + 1)
SWI2
```

Pierwsza instrukcja umieszcza w rejestrze A wartość n, a w rejestrze B wartość 1, natomiast druga instrukcja jest wywołaniem systemowym, czyli specjalnie traktowanym wywołaniem procedury, której kod znajduje się w pamięci ROM. Wywołanie systemowe implementujące instrukcję `MAlloc` jest w całości napisane w asemblerze procesora 6809 i najpierw próbuje przydzielić $2n$ bajtów na stercie. W przypadku, gdy nie ma na stercie miejsca, to usuwa ze sterty rekordy, które nie są już używane, a następnie ponawia próbę przydzielenia pamięci dla nowego rekordu. W przypadku drugiej porażki, wyświetla na standardowe wyjście komunikat `Out of memory` i kończy działanie programu. Podczas określania, które rekordy mogą być jeszcze używane, przyjęte jest następujące założenie: jeśli program używa rekordu przydzielonego przez instrukcję `MAlloc`, to trzyma jego adres w którymś rejestrze, na stosie (o adresie wyrównanym do dwóch bajtów) lub w innym takim rekordzie. Dlatego wykonywanie operacji arytmetycznych na adresach zwróconych przez `MAlloc` może prowadzić do dziwnego zachowania programu.

Instrukcje procesora 6809

Typ danych `MInstr` kodujący omawiane makroinstrukcje zawiera też konstruktor `MInstr` pozwalający osadzić dowolną instrukcję procesora 6809. Podobnie jak makroinstrukcje, instrukcje procesora 6809 wyrażamy jako elementy algebraicznego typu danych `Instr` zdefiniowanego w module `MPU6809`. Nie będziemy ich tutaj omawiać, ponieważ podane makroinstrukcje w zupełności wystarczają do rozwiązania zadania.

Zadanie

Zadanie jest dostępne w trzech wariantach, wartych odpowiednio 13, 20 i 30 punktów. W każdym z wariantów należy zaimplementować kompilator języka rozważanego w zadaniu na odpowiednio czwartą, piątą i szóstą pracownię. Wygenerowany przez kompilator kod powinien działać w następującym kontekście.

- Wartości zmiennych wejściowych znajdują się na stosie użytkownika w kolejności takiej, jak występują w programie.
- Na stosie systemowym znajduje się wskaźnik do kodu, który wyświetli wartość w akumulatorze i zakończy działanie programu. Oznacza to, że program po obliczeniu ostatecznej wartości, powinien ją umieścić w akumulatorze a następnie wykonać instrukcję `MRet`.

Dodatkowo można założyć, że wszystkie liczby pojawiające się podczas obliczania programu mieszczą się w 16 bitach, tj. w zakresie od -32768 do 32767, a system operacyjny w pamięci ROM poprawnie obsługuje dzielenie przez zero.

Skompilowane programy będą uruchamiane na maszynie, która ma tylko 64kB pamięci (z czego 16kB na stos użytkownika i 12kB na stos systemowy). Dla małych programów powinno to w zupełności wystarczyć, pod warunkiem, że nie korzystają one z głębokiej rekursji. Rozważ optymalizację rekursji ogonowej: zastanów się, kiedy można instrukcję wywołania funkcji `MCall` zamienić na instrukcję skoku `MJump`.

Wymogi formalne
Należy zgłosić pojedynczy plik o nazwie *ImięNazwisko*`Compiler.hs` gdzie za *Imię* i *Nazwisko* należy podstawić odpowiednio swoje imię i nazwisko zaczynające się wielką literą oraz bez znaków diakrytycznych. Plik ten powinien być napisany w Haskellu przy użyciu podzbioru *SafeHaskell* i powinien definiować moduł eksportujący funkcję `compile` tak jak opisano w szablonie do odpowiedniego wariantu zadania. Dodatkowo należy w komentarzu wewnątrz pliku zaznaczyć wybrany wariant zadania. **Rozwiązania nie spełniające wymogów formalnych nie będą oceniane!**

Uwaga
W serwisie SKOS umieszczono archiwum z szablonem rozwiązania i programem pozwalającym uruchamiać napisany kompilator. Podany szablon dostępny jest w dwóch wersjach: dla pracowni nr 5 i 6. Jeśli ktoś decyduje się rozwiązać zadanie w wersji 4, to należy użyć szablonu dla wersji 5 i zignorować konstrukcje języka dodane w zadaniu 5.

Dodatkowo w SKOSie umieszczony jest emulator pozwalający uruchamiać skompilowane programy. Na przykład, aby skompilować a następnie uruchomić program znajdujący się w pliku `example.pp5` wykonujemy następujące polecenia:

```
$ ./Comp5 example.pp5
$ ./emu6809 example.b09
```

Więcej informacji o zamieszczonych programach można znaleźć uruchamiając je z flagą `--help`.