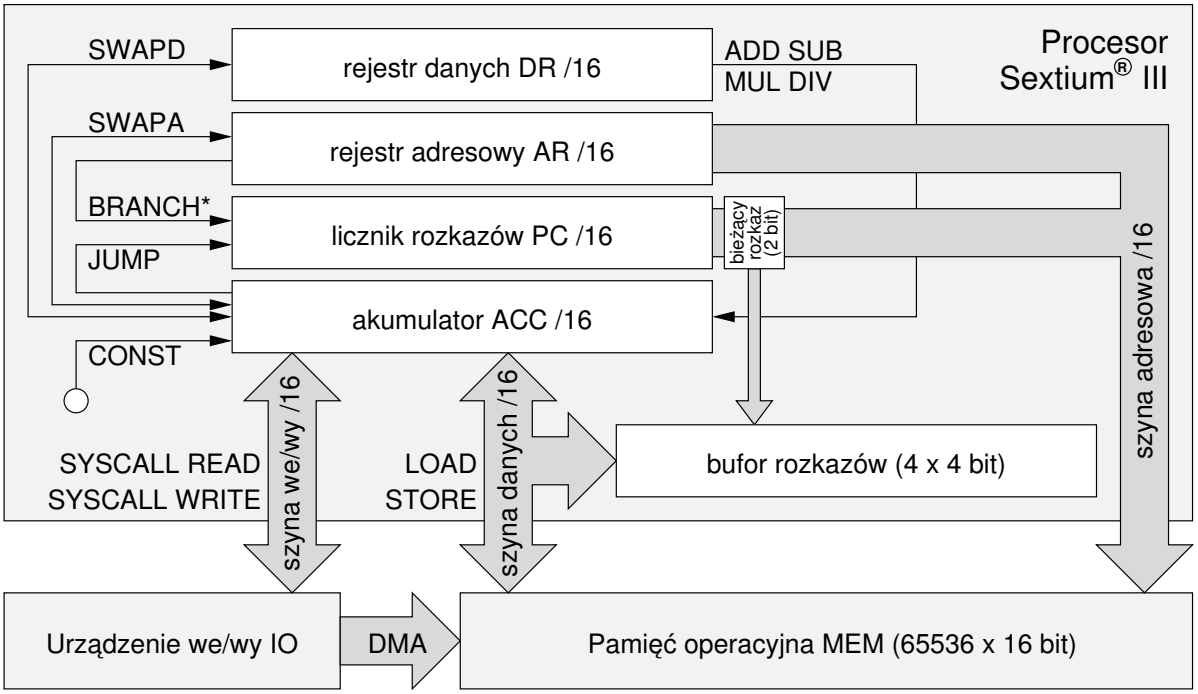
Metody programowania 2016

Lista zadań na pracownię nr 3

Termin zgłaszania w KNO: 4 maja 2016, godzina 6:00 AM CET

Procesor Sextium® III

Na poniższym rysunku przedstawiamy architekturę procesora Sextium® III zaprojektowanego specjalnie na potrzeby niniejszego zadania.[1]



gdzie:

ACC *accumulator*, akumulator: rejestr procesora, na którym są wykonywane operacje arytmetyczne i przesyłu danych.

DR *data register*, rejestr danych: pomocniczy rejestr przechowujący drugi argument operacji arytmetycznych.

AR *address register*, rejestr adresowy: rejestr, którego zawartość służy do adresowania pamięci w celu pobrania (zapisu) danych z (do) pamięci.

PC *program counter*, licznik rozkazów: rejestr, którego zawartość służy do adresowania pamięci w celu pobrania kolejnych rozkazów do wykonania.

MEM *memory*, pamięć operacyjna.

IO *input/output*, urządzenie wejścia/wyjścia.

DMA *direct memory access*: możliwość bezpośredniego zapisu do pamięci wprost z urządzenia wejścia/wyjścia bez udziału procesora.

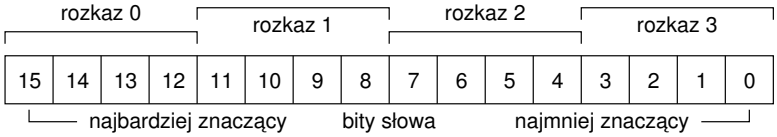
Poprzez szynę danych i szynę adresową do procesora jest podłączona pamięć MEM złożona z 65536 słów 16-bitowych, a poprzez szynę wejścia/wyjścia — urządzenie wejścia/wyjścia IO. Szyny danych, adresowa i we/wy są 16-bitowe, podobnie jak rejestry: akumulator ACC (akumulator jest w rzeczywistości 32-bitowy[2]), rejestr danych DR, rejestr adresowy AR i licznik rozkazów PC. Do przedstawiania zawartości pamięci i rejestrów procesora będziemy konsekwentnie używać zapisu szesnastkowego. Zawartość pojedynczego słowa będzie zatem opisana ciągiem czterech cyfr szesnastkowych. *Jednostką adresowania*, tj. najkrótszym ciągiem bitów, który można zaadresować w pamięci (*dawniej* zwanym *bajtem*) w procesorze Sextium® III jest słowo 16-bitowe, inaczej niż w większości współczesnych architektur, w których jednostką adresowania jest *oktet* (słowo

[1] Pierwsza wersja procesora Sextium® pojawiła się na zajęciach z Metod Programowania w roku akademickim 1999/2000 i została rok później zastąpiona poprawionym modelem Sextium® II. Procesor Sextium® III to unowocześniona konstrukcja wprowadzona do masowej produkcji w roku 2016.

[2] Procesor Sextium® III posiada pewne dodatkowe własności pominięte w niniejszej dokumentacji.

8-bitowe, *obecnie* zwane *bajtem*). Słowo 16-bitowe składa się z czterech *kwartetów* (zwanych *półbajtami* lub *niblami*). W naszej architekturze pojęcie bajtu nie występuje i jest ona szczególnie prosta, bo jednostka adresowania i słowo maszynowe są tej samej długości. Gdy programy dla procesora Sextium® III są przechowywane na nośnikach bajtowych (np. w komputerze PC), to zapisujemy je w formacie *big endian* (tj. bardziej znaczący bajt ma — inaczej niż np. w procesorze x86 — mniejszy adres). Rozkazy (jest ich 16)[3] zajmują po 4 bity i są pakowane po 4 w słowie maszyny. Położenie rozkazu w pamięci jest więc określone przez ciąg 18 bitów, który będziemy zapisywać w postaci $a_3a_2a_1a_0:n$, przy czym $a_3a_2a_1a_0$ jest adresem słowa maszyny (gdzie a_i to dowolne cyfry szesnastkowe), a $n \in \{0,\ldots,3\}$ oznacza położenie rozkazu w słowie.

Sposób pakowania rozkazów w słowie maszynowym jest przedstawiony na poniższym rysunku.



Dla wygody rozkazy mają swoje nazwy mnemoniczne. Zatem np. zamiast pisać „rozkaz `D`", będziemy pisać „rozkaz `DIV`". Dla procesora jednak rozkaz, to po prostu cztery bity (które my zapisujemy za pomocą pojedynczej cyfry szesnastkowej). Opis rozkazów jest przedstawiony w poniższej tablicy.

Rozkaz	Symb.	Definicja	Opis
`0`	`NOP`	nic nie rób	przejdź do wykonania następnego rozkazu
`1`	`SYSCALL`	*syscall*(ACC)	wywołaj funkcję systemową której kod jest umieszczony w akumulatorze
`2`	`LOAD`	MEM[AR] → ACC	załaduj słowo o adresie znajdującym się w rejestrze adresowym do akumulatora
`3`	`STORE`	ACC → MEM[AR]	zapisz zawartość akumulatora do słowa o adresie znajdującym się w rejestrze adresowym
`4`	`SWAPA`	ACC ↔ AR	zamień miejscami zawartość rejestru adresowego i akumulatora
`5`	`SWAPD`	ACC ↔ DR	zamień miejscami zawartość rejestru danych i akumulatora
`6`	`BRANCHZ`	if ACC = 0 then AR → PC	jeżeli akumulator zawiera liczbę 0, to zapisz do licznika instrukcji zawartość rejestru adresowego (skocz do instrukcji o adresie znajdującym się w rejestrze adresowym)
`7`	`BRANCHN`	if ACC < 0 then AR → PC	jeżeli akumulator zawiera liczbę ujemną, to zapisz do licznika instrukcji zawartość rejestru adresowego (skocz do instrukcji o adresie znajdującym się w rejestrze adresowym)
`8`	`JUMP`	ACC → PC	wpisz do licznika instrukcji zawartość akumulatora (skocz do instrukcji o adresie znajdującym się w akumulatorze)
`9`	`CONST`	MEM[PC++] → ACC	zapisz słowo znajdujące się w komórce pamięci o adresie wskazywanym przez bieżącą zawartość licznika rozkazów do akumulatora
`A`	`ADD`	ACC + DR → ACC	dodaj do akumulatora zawartość rejestru danych
`B`	`SUB`	ACC − DR → ACC	odejmij od akumulatora zawartość rejestru danych
`C`	`MUL`	ACC × DR → ACC	pomnóż zawartość akumulatora przez zawartość rejestru danych
`D`	`DIV`	ACC / DR → ACC	podziel zawartość akumulatora przez zawartość rejestru danych
`E`			nieudokumentowane[4]
`F`			

Rozkaz o kodzie 1 (`SYSCALL`) powoduje wywołanie funkcji systemowej której kod jest umieszczony w akumulatorze (a parametry — jeśli są potrzebne — w innych rejestrach). Wynik działania funkcji systemowej jest umieszczony w akumulatorze.

[3] Spośród których przedstawiamy 14 (zob. poprzedni przypis).
[4] Zob. poprzedni przypis.

Kody dostępnych funkcji systemowych są przedstawione w poniższej tabeli.

ACC	Symb.	Definicja	Opis
0	HALT	zatrzymaj	zakończ program, przekaż sterowanie do urządzenia wejścia/wyjścia
1	READ	IO → ACC	wczytaj słowo z urządzenia wejściowego do akumulatora
2	WRITE	DR → IO	prześlij zawartość rejestru danych do urządzenia wyjściowego

Cykl rozkazowy procesora polega na pobieraniu, dekodowaniu i wykonywaniu rozkazów (gdzie A++ oznacza zwiększenie zawartości rejestru A o jeden):

```
wyzeruj rejestry ACC, PC, AR i DR i opróżnij bufor rozkazów
loop
    if bufor rozkazów jest pusty then
        pobierz do bufora rozkazów słowo z pamięci o adresie zawartym w PC
        PC++
    end if
    R ← odkoduj następny rozkaz
    if R = NOP then
        continue
    else if R = SYSCALL then
        przekaż sterowanie do BIOS-u maszyny
        if wystąpił błąd funkcji systemowej or wywołano funkcję HALT then
            break
        end if
    else if R = LOAD, STORE, SWAPA, SWAPD then
        prześlij zawartość odpowiednich rejestrów lub słów pamięci
    else if (R = BRANCHZ and ACC = 0) or (R = BRANCHN and ACC < 0) then
        zapisz do PC zawartość AR
        opróżnij bufor rozkazów
    else if R = JUMP then
        zapisz do PC zawartość ACC
        opróżnij bufor rozkazów
    else if R = CONST then
        załaduj słowo z pamięci o adresie zawartym w PC do ACC
        PC++
    else if R = ADD, SUB, MUL, DIV then
        if R = DIV i DR = 0 then
            break
        end if
        wykonaj odpowiednią operację arytmetyczną
    end if
end loop
```

Wczytane z pamięci słowo zawierające cztery rozkazy jest przechowywane w specjalnym *buforze rozkazów*. W celu pobrania rozkazów procesor odwołuje się do pamięci przeważnie raz na cztery cykle rozkazowe, chyba że wykonuje instrukcję skoku. Wówczas bufor rozkazów jest opróżniany nawet jeśli rozkaz skoku nie jest ostatnim rozkazem w słowie. Po wykonaniu skoku, tj. po zmianie zawartości licznika rozkazów i pobraniu nowego słowa do bufora rozkazów kolejnym interpretowanym rozkazem jest rozkaz o numerze 0 w słowie. Można zatem skoczyć jedynie do rozkazu, który zajmuje cztery najbardziej znaczące bity słowa (jeśli sprawia to jakieś problemy, można poprzednie słowo wypełnić rozkazami NOP w celu wyrównania wskazanego rozkazu do granicy słowa). Rozkaz CONST służy do załadowania stałej do akumulatora. Stała jest umieszczana w słowie następującym bezpośrednio po słowie zawierającym ten rozkaz (lub w następnych słowach, jeżeli w jednym słowie znajduje się więcej niż jeden rozkaz CONST).

Podczas wykonywania instrukcji MUL, DIV i BRANCHN oraz funkcji systemowych READ i WRITE zawartości rejestrów są interpretowane jako liczby całkowite ze znakiem z przedziału $-2^{15} \div (2^{15} - 1)$ w zapisie uzupełnieniowym do dwóch, tj. zmiana znaku liczby polega na zanegowaniu wszystkich bitów liczby i dodaniu jedynki. Słowa od 0000 do 7FFF reprezentują liczby nieujemne 0 do 32767, słowa 8000 do FFFF zaś liczby −32768 do −1. Nadmiar i niedomiar występujący podczas wykonania operacji ADD, SUB i MUL oraz funkcji systemowej READ nie jest w żaden sposób sygnalizowany.

Po uruchomieniu maszyny program dla procesora jest wpisywany do pamięci wprost z urządzenia wejścia/wyjścia (poprzez tzw. szynę DMA). W tym czasie procesor nie pracuje. Następnie urządzenie wejścia/wyjścia wysyła sygnał do procesora, który zeruje zawartość wszystkich swoich rejestrów i opróżnia bufor rozkazów, a następnie rozpoczyna wykonywanie cyklu rozkazowego. Procesor przekazuje sterowanie na powrót do urządzenia wejścia/wyjścia po wykonaniu rozkazu SYSCALL z parametrem HALT, jeśli wywołanie funkcji systemowej się nie powiodło lub w razie próby dzielenia przez zero. Pracę procesora

może również przerwać urządzenie wejścia/wyjścia. Po uruchomieniu licznik rozkazów ma wartość `0000`, zatem procesor rozpoczyna działanie od wykonania rozkazów znajdujących się w zerowym słowie pamięci.

Obecnie podamy kilka przykładów programów zapisanych w kodzie maszynowym procesora Sextium® III i opiszemy jak można łatwo z nimi eksperymentować. Następująca para słów: `59a0 0001` zawiera ciąg rozkazów `SWAPD`, `CONST`, `ADD`, `NOP`, po którym następuje liczba 1[5]. Wykonanie tych rozkazów powoduje zwiększenie zawartości akumulatora o jeden. Podobnie ciąg sześciu słów:

```
959C 0004 0005 5945 FFFA 3000
```

który symbolicznie zapiszemy następująco:

```
0000: CONST SWAPD CONST MUL
0001: 0004
0002: 0005
0003: SWAPD CONST SWAPA SWAPD
0004: fffa
0005: STORE NOP NOP NOP
```

powoduje załadowanie do rejestru danych liczby 4, do akumulatora liczby 5, przemnożenie zawartości akumulatora przez zawartość rejestru danych i zapisanie wyniku (liczby 20) do komórki pamięci o adresie `fffa`.

Za pomocą najprostszych narzędzi możemy łatwo manipulować takimi programami. Standardowe polecenia `xxd` i `hexdump (hd)` zamieniają ciągi cyfr szesnastkowych na pliki binarne i wypisują zawartość plików binarnych w postaci szesnastkowej. Np. polecenie

```
echo "9491 0006 0001 6a59 0000 8000 9191 0002 0000" | xxd -p -r > sum.sextium
```

spowoduje utworzenie pliku binarnego programu, który wczytuje z urządzenia wejścia/wyjścia ciąg liczb aż napotka liczbę 0, a wówczas wypisuje sumę wczytanych liczb i kończy działanie. Analizowanie programu zapisanego szesnastkowo jest żmudne, dlatego wolimy zapisać ten program w bardziej symbolicznej notacji, podobnej do asemblera:[6]

```
0000: CONST SWAPA CONST SYSCALL     ; read(ACC)
0001: 0006                          ;    adres skoku, gdy wczytano 0
0002: 0001                          ;    kod syscall READ
0003: BRANCHZ ADD SWAPD CONST       ; if (ACC=0) goto 0006
0004: 0000                          ;    adres skoku bezwarunkowego
0005: JUMP NOP NOP NOP              ; goto 0000
0006: CONST SYSCALL CONST SYSCALL   ; write(ACC); halt();
0007: 0002                          ;    kod syscall WRITE
0008: 0000                          ;    kod syscall HALT
```

Za pomocą prostego narzędzia `sed` możemy automatycznie zastąpić nazwy instrukcji procesora ich kodami i pominąć numery wierszy (ciągi cyfr szesnastkowych i białych znaków aż do znaku „:”), które bardzo ułatwiają orientację w kodzie oraz komentarze (ciągi znaków od znaku „;” do końca wiersza). Przykład odpowiedniego skryptu jest dostępny w serwisie KNO na stronie zajęć razem z emulatorem procesora Sextium® III. Skonwertowany do postaci binarnej program możemy uruchomić poleceniem `sextium` z parametrem będącym nazwą pliku binarnego.

Korzystając ze wspomnianych wyżej elementarnych narzędzi możemy eksperymentować nawet z „dużymi” programami. Rozważmy np. algorytm Euklidesa wyliczający największy wspólny podzielnik dwóch liczb:

read(x)
read(y)
while $y \neq 0$ **do**
    $t \leftarrow x \bmod y$
    $x \leftarrow y$
    $y \leftarrow t$
end while
write(x)

(z powodu zaokrąglania podczas dzielenia w kierunku zera algorytm daje spodziewane wyniki jedynie dla liczb nieujemnych).

[5] Dzięki przyjęciu konwencji *big endian* powyższy ciąg dwóch słów możemy zapisać jako następujący ciąg czterech bajtów: `59 a0 00 01`. W konwencji *little endian* byłoby to `a0 59 01 00`.

[6] Prawdziwe asemblery to tzw. języki *adresów* symbolicznych, które poza symbolicznym przedstawianiem kodów instrukcji pozwalają także oznaczać symbolicznie (tj. za pomocą identyfikatorów) adresy w segmencie kodu (etykiety) i segmencie danych (zmienne statyczne).

Powyższy algorytm możemy zakodować w „asemblerze” procesora Sextium® III następująco:

```
0000: CONST SWAPA CONST SYSCALL   ; read(x)
0001: 001a                        ;    adres x
0002: 0001                        ;    kod syscall READ
0003: STORE CONST SWAPA CONST     ; read(y)
0004: 001b                        ;    adres y
0005: 0001                        ;    kod syscall READ
0006: SYSCALL STORE SWAPD NOP     ; DR := y
                                  ; etykieta dalej = 0007
0007: CONST SWAPA SWAPD BRANCHZ   ; if y=0 goto koniec
0008: 0015                        ;    adres etykiety koniec
0009: SWAPD CONST SWAPA LOAD      ; ACC := x
000a: 001a                        ;    adres x
000b: DIV CONST SWAPD SHIFT       ; ACC := x mod y
000c: fff0                        ;    stała -16
000d: SWAPD CONST SWAPA LOAD      ; DR := y
000e: 001b                        ;    adres y
000f: SWAPD STORE CONST SWAPA     ; y := ACC; x := DR
0010: 001a                        ;    adres x
0011: SWAPD STORE CONST SWAPA     ; DR := y
0012: 001b                        ;    adres y
0013: LOAD SWAPD CONST JUMP       ; goto dalej
0014: 0007                        ;   adres etykiety dalej
                                  ; etykieta koniec = 0015
0015: CONST SWAPA LOAD SWAPD      ; DR := x
0016: 001a                        ;    adres x
0017: CONST SYSCALL CONST SYSCALL ; write(x); halt()
0018: 0002                        ;    kod syscall WRITE
0019: 0000                        ;    kod syscall HALT
                                  ; Sekcja danych:
001a:                             ; zmienna x
001b:                             ; zmienna y
```

Program składa się z 26 słów. Zmienne x i y przechowujemy w słowach o adresach `001a` i `001b` następujących w pamięci bezpośrednio po instrukcjach programu. Po automatycznym zakodowaniu rozkazów i usunięciu dodatkowego tekstu otrzymujemy następującą postać szesnastkową:

```
9491 001a 0001 3949 001b 0001 1350 9456 0015 5942 001a d95e fff0 5942 001b 5394
001a 5394 001b 2598 0007 9425 001a 9191 0002 0000
```

Studenci, którzy chcieliby poćwiczyć programowanie w języku maszynowym procesora Sextium® III mogą[7] napisać następujące programy:

1. Program wczytujący liczbę całkowitą i wypisujący liczbę jej nietrywialnych (różnych od 1 i od niej samej) podzielników.
2. Program wczytujący trzy liczby x, y i z i wyliczający $x^y \bmod z$.
3. Program wczytujący dwie liczby x i y i wyliczający $x! \bmod y$.
4. Program wczytujący sześć liczb całkowitych opisujących dwie chwile w ciągu doby w formacie godzina-minuta-sekunda i wypisujący liczbę sekund dzielących obie chwile (ujemną, jeśli druga chwila jest wcześniejsza niż pierwsza).
5. Program wczytujący liczbę całkowitą będącą numerem roku i wypisujący liczbę dni tego roku, tj. 365 lub, dla lat przestępnych, 366 (rok jest przestępny, jeśli jego numer dzieli się przez 4 i nie dzieli się przez 100 lub dzieli się przez 400).
6. Program wczytujący sześć liczb całkowitych opisujących dwie daty w formacie dzień-miesiąc-rok i wypisujący liczbę dni dzielących te daty (ujemną, jeśli druga data jest wcześniejsza niż pierwsza).

Pewna wprawa w programowaniu procesora Sextium® III przyda się do rozwiązania bieżącego zadania, ale nie zamierzamy rozwijać narzędzi służących do programowania w języku maszynowym. Zamiast tego opisujemy poniżej język znacznie wyższego poziomu, w którym programuje się znacznie wygodniej. Zadanie na pracownię polega na napisaniu w Prologu skrośnego[8] kompilatora tego języka generującego kod maszynowy procesora Sextium® III.

[7] Załączone ćwiczenia nie są częścią zadania na pracownię.

[8] Kompilator *skrośny* generuje kod wynikowy na inną maszynę niż ta, na której jest wykonywany.

Język Algol 16

Tokenami języka Algol 16 są:

- operatory: `+ - * < <= > >= = <> := ;`
- symbole przestankowe: `, ( )`
- słowa kluczowe: `and begin call div do done else end fi if local mod not or procedure program read return then value while write`
- identyfikatory: niepuste ciągi złożone z małych i wielkich liter ASCII, cyfr `0-9` i znaków `_ '` (podkreślenie, apostrof) zaczynające się literą i różne od słów kluczowych,
- literały całkowitoliczbowe: niepuste ciągi cyfr `0-9`.

Podczas analizy leksykalnej przyjmuje się zasadę zachłanności. Tokeny rozdziela się białymi znakami: spacji, tabulacji i nowego wiersza. Równoważne białym znakom są też komentarze, tj. ciągi znaków zaczynające się znakami `(*` po których następuje dowolny ciąg znaków nie zawierający podciągu `*)` i zakończonych znakami `*)`.

Struktura składniowa języka Algol 16 jest następująca:

⟨program⟩ ::= `program` ⟨identyfikator⟩ ⟨blok⟩

⟨blok⟩ ::= ⟨deklaracje⟩ `begin` ⟨instrukcja złożona⟩ `end`

⟨deklaracje⟩ ::= ⟨puste⟩ | ⟨deklaracje⟩ ⟨deklaracja⟩

⟨deklaracja⟩ ::= ⟨deklarator⟩ | ⟨procedura⟩

⟨deklarator⟩ ::= `local` ⟨zmienne⟩

⟨zmienne⟩ ::= ⟨zmienna⟩ | ⟨zmienne⟩ `,` ⟨zmienna⟩

⟨zmienna⟩ ::= ⟨identyfikator⟩

⟨procedura⟩ ::= `procedure` ⟨nazwa procedury⟩ `(` ⟨argumenty formalne⟩ `)` ⟨blok⟩

⟨nazwa procedury⟩ ::= ⟨identyfikator⟩

⟨argumenty formalne⟩ ::= ⟨puste⟩ | ⟨ciąg argumentów formalnych⟩

⟨ciąg argumentów formalnych⟩ ::= ⟨argument formalny⟩ | ⟨ciąg argumentów formalnych⟩ `,` ⟨argument formalny⟩

⟨argument formalny⟩ ::= ⟨zmienna⟩ | `value` ⟨zmienna⟩

⟨instrukcja złożona⟩ ::= ⟨instrukcja⟩ | ⟨instrukcja złożona⟩ `;` ⟨instrukcja⟩

⟨instrukcja⟩ ::= ⟨zmienna⟩ `:=` ⟨wyrażenie arytmetyczne⟩
| `if` ⟨wyrażenie logiczne⟩ `then` ⟨instrukcja złożona⟩ `fi`
| `if` ⟨wyrażenie logiczne⟩ `then` ⟨instrukcja złożona⟩ `else` ⟨instrukcja złożona⟩ `fi`
| `while` ⟨wyrażenie logiczne⟩ `do` ⟨instrukcja złożona⟩ `done`
| `call` ⟨wywołanie procedury⟩
| `return` ⟨wyrażenie arytmetyczne⟩
| `read` ⟨zmienna⟩
| `write` ⟨wyrażenie arytmetyczne⟩

⟨wyrażenie arytmetyczne⟩ ::= ⟨składnik⟩ | ⟨wyrażenie arytmetyczne⟩ ⟨operator addytywny⟩ ⟨składnik⟩

⟨operator addytywny⟩ ::= `+` | `-`

⟨składnik⟩ ::= ⟨czynnik⟩ | ⟨składnik⟩ ⟨operator multiplikatywny⟩ ⟨czynnik⟩

⟨operator multiplikatywny⟩ ::= `*` | `div` | `mod`

⟨czynnik⟩ ::= ⟨wyrażenie proste⟩ | `-` ⟨wyrażenie proste⟩

⟨wyrażenie proste⟩ ::= ⟨wyrażenie atomowe⟩ | `(` ⟨wyrażenie arytmetryczne⟩ `)`

⟨wyrażenie atomowe⟩ ::= ⟨zmienna⟩ | ⟨wywołanie procedury⟩ | ⟨literał całkowitoliczbowy⟩

⟨wywołanie procedury⟩ ::= ⟨nazwa procedury⟩ `(` ⟨argumenty faktyczne⟩ `)`

⟨argumenty faktyczne⟩ ::= ⟨puste⟩ | ⟨ciąg argumentów faktycznych⟩

⟨ciąg argumentów faktycznych⟩ ::= ⟨argument faktyczny⟩ | ⟨ciąg argumentów faktycznych⟩ `,` ⟨argument faktyczny⟩

⟨argument faktyczny⟩ ::= ⟨wyrażenie arytmetyczne⟩

⟨wyrażenie logiczne⟩ ::= ⟨koniunkcja⟩ | ⟨wyrażenie logiczne⟩ `or` ⟨koniunkcja⟩

⟨koniunkcja⟩ ::= ⟨warunek⟩ | ⟨koniunkcja⟩ `and` ⟨warunek⟩

⟨warunek⟩ ::= ⟨wyrażenie relacyjne⟩ | `not` ⟨wyrażenie relacyjne⟩

⟨wyrażenie relacyjne⟩ ::= ⟨wyrażenie arytmetyczne⟩ ⟨operator relacyjny⟩ ⟨wyrażenie arytmetyczne⟩
| `(` ⟨wyrażenie logiczne⟩ `)`

⟨operator relacyjny⟩ ::= `<` | `<=` | `>` | `>=` | `=` | `<>`

⟨puste⟩ ::=

Jedynym typem danych w Algolu 16 są liczby całkowite. Występują w nim jedynie procedury funkcyjne zwracające wartość liczbową. Jeśli wynik działania procedury jest nieistotny, to można ją wywołać w instrukcji procedury rozpoczynającej się słowem kluczowym `call`. Wynik zwrócony przez tę procedurę zostanie wówczas odrzucony.

Procedura składa się z nagłówka, ciągu deklaracji lokalnych zmiennych i procedur oraz instrukcji złożonej (tj. ciągu instrukcji) ujętego w nawiasy `begin` i `end`. Procedury mogą być wywoływane rekurencyjnie. Cały program jest zbudowany podobnie do procedury, tylko w miejscu nagłówka procedury posiada słowo kluczowe `program` po którym następuje nazwa programu. Nazwa programu jest jedynie komentarzem i nie można się na nią powoływać w treści programu.

W nagłówku procedury jest wymieniona jej nazwa oraz parametry. Domyślnie parametry są przekazywane przez nazwę (*call by name*, tak jak w Algolu 60). Jeśli przed nazwą parametru formalnego umieszczono słowo kluczowe `value`, to odpowiedni parametr jest przekazywany przez wartość (*call by value*, tak jak w Algolu 60 i większości współczesnych języków programowania). Semantyka tych sposobów wywołania jest taka, jak opisana w punkcie 4.7.3 raportu [1].

W treści procedury można się odwoływać do nazw zmiennych i procedur zadeklarowanych nielokalnie. Obowiązuje wówczas zasada statycznego (leksykalnego) wiązania nazwy z deklaracją, tj. bierze się pod uwagę statyczny (leksykalny) zasięg deklaracji (tak jak w Algolu 60 i większości współczesnych języków programowania).

Wykonanie instrukcji `return` *e* powoduje obliczenie wyrażenia *e*, zakończenie wykonania procedury i przekazanie stronie wywołującej wyliczonej wartości wyrażenia *e* jako wyniku wykonania procedury. Jeśli ostatnią instrukcją wykonaną w treści procedury nie jest instrukcja `return`, wówczas wynikiem tej procedury jest liczba 0. Wykonanie instrukcji `write` *e* powoduje obliczenie wyrażenia *e* i wysłanie jego wartości do urządzenia wejścia/wyjścia. Instrukcja `read` *x* powoduje wczytanie liczby z urządzenia wejścia/wyjścia i przesłanie jej do zmiennej *x*.

Poniżej podajemy przykłady kilku poprawnych programów w języku Algol 16.

```
program Suma
   local x, s
begin
   s := 0;
   read x;
   while x <> 0 do
      s := s + x;
      read x
   done;
   write s
end
```

```
program gcd
   local x, y, t
begin
   read x;
   read y;
   while y <> 0 do
      t := x mod y;
      y := x;
      x := t
   done;
   write y
end
```

```
program Piec_Dziesiec_Pietnascie
   procedure f (x)
      procedure one ()
      begin
         write 5;
         return 1
      end
   begin
      if x > 0
      then
         f (x - one())
      fi
   end
begin
   f (5)
end
```

```
program Jensen
   procedure sum (expr, index)
      local result
   begin
      result := 0;
      while index > 0 do
         result := result + expr;
         index := index - 1
      done;
      return result
   end
   local n
begin
   n := 5;
   write sum (n * n + 1, n)
end
```

Program `Suma` wczytuje liczby aż do napotkania liczby 0 i wypisuje sumę wczytanych liczb. Program gcd oblicza największy wspólny podzielnik wczytanych liczb. Program `Piec_Dziesiec_Pietnascie` wypisuje piętnaście razy liczbę 5. Gdyby parametr `x` funkcji `f` był przekazywany przez wartość, wówczas program wypisałby liczbę 5 pięć razy. Program `Jensen` to przykładowa implementacja tzw. *wynalazku Jensena*. Program wypisuje wartość $\sum_{n=1}^{5}(n^2+1)$.

Literatura

[1] Revised Report on the Algorithmic Language Algol 60, Peter Naur, ed., *Comm. ACM* **6**(1):1–17 Jan. 1963.

[2] Niklaus Wirth, *Algorytmy + struktury danych = programy*, WNT, 2004.

Zadanie na pracownię

Zdefiniuj w Prologu predykat `algol16(+Source,-SextiumBin)` który dla podanej listy `Source` liczb całkowitych będących kodami ASCII kolejnych znaków programu źródłowego w języku będącym podzbiorem języka Algol 16 tworzy listę `SextiumBin` liczb całkowitych z przedziału 0–65535 reprezentujących zawartość kolejnych słów pamięci programu będącego tłumaczeniem podanego programu źródłowego na kod maszynowy procesora Sextium® III. W pełnej wersji zadania do wyboru są dwa podzbiory języka: 1) język bez możliwości zagnieżdżania procedur (procedury można deklarować jedynie w bloku programu; w blokach procedur są możliwe jedynie deklaracje lokalnych zmiennych), 2) język bez możliwości przekazywania parametrów do procedur przez nazwę. Możliwe jest także zaimplementowanie mniejszego podzbioru języka, jednak wówczas można uzyskać mniej niż 40 punktów. Najuboższa wersja języka jest zupełnie pozbawiona procedur. W bloku programu można deklarować jedynie zmienne statyczne. Za zaimplementowanie tak okrojonej wersji języka można otrzymać jedynie 22 punkty.[9] Cennik za zaimplementowanie dalszych własności języka jest następujący:

— nierekurencyjne procedury bez parametrów z lokalnymi zmiennymi 3

— możliwość rekurencyjnego wywoływania procedur 3

— możliwość przekazywania parametrów do procedur przez wartość 2

— możliwość przekazywania parametrów do procedur przez nazwę 10

— możliwość zagnieżdżania procedur 10

Możliwe jest także zaimplementowanie pełnego języka Algol 16. Za przedstawienie takiego rozwiązania student otrzyma 10 punktów bonusowych, czyli razem 50 punktów.

Program w Prologu należy umieścić w pojedynczym pliku o nazwie `Imie_Nazwisko.pl` (gdzie Imie i Nazwisko to imię i nazwisko zgłaszającego studenta zapisane bez znaków diakrytycznych). Na początku pliku powinien się znajdować komentarz zawierający co najmniej imię i nazwisko autora programu oraz informację o zaimplementowanej wersji języka.

Niniejszy plik wraz z emulatorem procesora Sextium® III i przykładowym programami jest dostępny w serwisie KNO na stronie zajęć.

[9] Wybranie tej wersji zadania jest nieco ryzykowne: aby zaliczyć pracownię trzeba otrzymać co najmniej 20 punktów za to zadanie, można więc stracić co najwyżej dwa punkty za usterki.