Metody programowania 2017

Lista zadań na pracownię nr 3

Celem poprzednich list zadań było napisanie programów badających spełnialność zbioru klauzul rachunku zdań. Choć dla niektórych danych takie programy działają bardzo długo, to znajdują one wiele praktycznych zastosowań, ponieważ wiele interesujących problemów można sprowadzić do sprawdzenia, czy zadany zbiór klauzul jest spełnialny. Na tej liście zadań zajmiemy się jednym z takich problemów, czyli weryfikowaniem poprawności układów cyfrowych.

Język HDML

Układy cyfrowe projektuje się za pomocą specjalnych języków opisu sprzętu (HDL, ang. Hardware Description Language), np. *Verilog* albo *VHDL*. Z uwagi na skomplikowaną semantykę tych języków, zajmiemy się znacznie prostszym językiem HDML wymyślonym na potrzeby tego zadania. Poniżej przedstawiamy definicję jego składni.

Białymi znakami w języku HDML są znaki o kodach ASCII 9–13 oraz 32, czyli spacja oraz znaki tabulacji (pionowej i poziomej), nowego wiersza, powrotu karetki i zakończenia strony. Sekwencje znaków `(*` oraz `*)` odpowiednio rozpoczynają i kończą komentarz, a pomiędzy nimi może znajdować się dowolna sekwencja znaków, nie zawierająca sekwencji `*)`, która nie zamyka zagnieżdżonego komentarza. Komentarze można zagnieżdżać.

Składnia

Tokenami języka HDML są:

- słowa kluczowe:
  `def` `else` `if` `in` `let` `then` `_` (podkreślenie)
- operatory i znaki przestankowe:
  `(` `)` `[` `]` `..` `,` `=` `<>` `<` `>` `<=` `>=` `^` `|` `+` `-`
  `&` `*` `/` `%` `@` `#` `~`
- identyfikatory: niepuste identyfikatory składające się z małych i wielkich liter ASCII, cyfr `0-9` i znaków `_ '` (podkreślenie i apostrof) zaczynające się literą lub podkreśleniem i różne od słów kluczowych,
- literały całkowitoliczbowe: niepuste ciągi cyfr `0-9`.

Podczas analizy leksykalnej przyjmuje się zasadę zachłanności. Tokeny rozdzielane są dodatkowo przez białe znaki oraz komentarze.

Programy w języku HDML opisane są następującą gramatyką:

⟨program⟩	::=	⟨definicje⟩
⟨definicje⟩	::=	⟨puste⟩ | ⟨definicja⟩ ⟨definicje⟩
⟨definicja⟩	::=	`def` ⟨identyfikator⟩ `(` ⟨wzorzec⟩ `)` `=` ⟨wyrażenie⟩
⟨wzorzec⟩	::=	`_` | ⟨zmienna⟩ | `(` ⟨wzorzec⟩ `)` | ⟨wzorzec⟩ `,` ⟨wzorzec⟩
⟨wyrażenie⟩	::=	`if` ⟨wyrażenie⟩ `then` ⟨wyrażenie⟩ `else` ⟨wyrażenie⟩
	|	`let` ⟨wzorzec⟩ `=` ⟨wyrażenie⟩ `in` ⟨wyrażenie⟩
	|	⟨wyrażenie op.⟩
⟨wyrażenie op.⟩	::=	⟨wyrażenie op.⟩ ⟨operator binarny⟩ ⟨wyrażenie op.⟩
	|	⟨operator unarny⟩ ⟨wyrażenie op.⟩
	|	⟨wyrażenie proste⟩
⟨operator binarny⟩	::=	`,` | `=` | `<>` | `<` | `>` | `<=` | `>=` | `^` | `|` | `+` | `-` | `&` | `*` | `/` | `%` | `@`

⟨operator unarny⟩ ::= `-` | `#` | `~`
⟨wyrażenie proste⟩ ::= `(` ⟨wyrażenie⟩ `)`
| ⟨wybór bitu⟩
| ⟨wybór bitów⟩
| ⟨wyrażenie atomowe⟩
⟨wybór bitu⟩ ::= ⟨wyrażenie proste⟩ `[` ⟨wyrażenie⟩ `]`
⟨wybór bitów⟩ ::= ⟨wyrażenie proste⟩ `[` ⟨wyrażenie⟩ `..` ⟨wyrażenie⟩ `]`
⟨wyrażenie atomowe⟩ ::= ⟨zmienna⟩
| ⟨wywołanie funkcji⟩
| ⟨literał całkowitoliczbowy⟩
| ⟨pusty wektor⟩
| ⟨pojedynczy bit⟩
⟨zmienna⟩ ::= ⟨identyfikator⟩
⟨wywołanie funkcji⟩ ::= ⟨identyfikator⟩ `(` ⟨wyrażenie⟩ `)`
⟨pusty wektor⟩ ::= `[ ]`
⟨pojedynczy bit⟩ ::= `[` ⟨wyrażenie⟩ `]`
⟨puste⟩ ::=

Niestety podana gramatyka nie jest jednoznaczna. Aby uczynić ją jednoznaczną przyjmujemy priorytety i łączność operatorów binarnych zgodnie z poniższą tabelką.

operatory	łączność	priorytet
`&` `*` `/` `%`	w lewo	5
`|` `^` `+` `-`	w lewo	4
`@`	w prawo	3
`=` `<>` `<` `>` `<=` `>=`	brak	2
`,`	w prawo	1

Dodatkowo zakładamy, że operatory unarne wiążą silniej niż wszystkie operatory binarne, a w gramatyce wzorców `,` (przecinek) wiąże w prawo.

Abstrakcyjne drzewo rozbioru

Programy w języku HDML będziemy reprezentować za pomocą termów prologowych przyjmując następujące konwencje.

Pozycja w kodzie źródłowym Większość wierzchołków abstrakcyjnego drzewa rozbioru zawiera informację na temat odpowiadającej im pozycji w kodzie źródłowym. Pozycje reprezentujemy na jeden z trzech sposobów:

`file(Path, Line, LinePos, CharNo, Length)`: gdzie `Path` oznacza ścieżkę do pliku, `Line` jest numerem wiersza (licząc od 1), `LinePos` jest numerem znaku w wierszu (licząc od 1), `CharNo` jest numerem znaku w pliku (licząc od 0), a `Length` jest długością wskazywanego miejsca.

`file(Path, Line, LinePos, CharNo)`: Podobnie jak wyżej, ale bez podawania długości.

`no`: Atom `no` oznacza brak informacji na temat pozycji w kodzie źródłowym.

Zmienna Zmienne reprezentujemy jako zwykłe atomy prologowe. Możesz użyć standardowego predykatu `atom_codes/2` do zamiany ciągu znaków na atom.

Operator Operatory unarne i binarne (różne od `,`) reprezentujemy jako zwykłe atomy prologowe, np. operator `<` jest reprezentowany jako term `'<'`.

Wzorzec Wzorce reprezentujemy jako termy w następujący sposób (`Pos` oznacza pozycje wzorca w kodzie źródłowym):

`wildcard(Pos)`: wzorzec postaci _;

`var(Pos, X)`: wzorzec będący zmienną reprezentowaną przez atom `X`.

`pair(Pos, P1, P2)`: wzorzec postaci *P1*,*P2*.

Wzorce postaci (*P*) są reprezentowane bezpośrednio przez *P*: jeśli program jest już w postaci drzewa, to nawiasy nie są potrzebne.

Wyrażenie Wyrażenia reprezentujemy jako termy w następujący sposób (`Pos` oznacza pozycje wyrażenia w kodzie źródłowym):

`if(Pos, E1, E2, E3)`: wyrażenie `if` *E1* `then` *E2* `else` *E3*

`let(Pos, P, E1, E2)`: wyrażenie `let` *P* `=` *E1* `in` *E2*

`op(Pos, Op, E)`: wyrażenie ⊕*E*, gdzie `Op` jest atomem reprezentującym operator unarny ⊕.

`op(Pos, Op, E1, E2)`: wyrażenie *E1* ⊕ *E2*, gdzie `Op` jest atomem reprezentującym operator binarny ⊕ różny od przecinka (patrz następny punkt).

`pair(Pos, E1, E2)`: wyrażenie *E1* `,` *E2*

`bitsel(Pos, E1, E2)`: wyrażenie *E1* `[` *E2* `]`

`bitsel(Pos, E1, E2, E3)`: wyrażenie *E1* `[` *E2* `..` *E3* `]`

`call(Pos, Name, E)`: wywołanie funkcji o nazwie reprezentowanej przez atom `Name` i parametrze `E`.

`var(Pos, X)`: wyrażenie będące zmienną o nazwie reprezentowanej przez atom `X`.

`num(Pos, N)`: numerał całkowitoliczbowy o wartości `N`

`empty(Pos)`: wyrażenie `[ ]`

`bit(Pos, E)`: wyrażenie `[` *E* `]`

Podobnie jak w przypadku wzorców, nie ma specjalnej konstrukcji reprezentującej nawiasy.

Definicja Definicje funkcji reprezentujemy jako term postaci `def(Name, P, E)`, gdzie `Name` jest atomem reprezentującym nazwę funkcji, `P` reprezentuje wzorzec argumentu formalnego funkcji, a `E` reprezentuje wyrażenie będące ciałem funkcji.

Program Programy reprezentujemy jako zwykłe listy prologowe, których elementami są termy reprezentujące definicje.

Na przykład poniższy program

```
def half_adder(A, B) =
  A & B, A ^ B
```

jest reprezentowany przez term

```
[ def(half_adder, pair(no, var(no, 'A'), var(no, 'B')),
    pair(no, op(no, '&', var(no, 'A'), var(no, 'B')),
             op(no, '^', var(no, 'A'), var(no, 'B')))) ]
```

Dla poprawienia czytelności program nie zawiera informacji o pozycji poszczególnych elementów. Program z pełną informacją na temat pozycji wygląda:

```
[ def(half_adder, pair(file('example.hdml', 1, 16, 15, 4),
      var(file('example.hdml', 1, 16, 15, 1), 'A'),
      var(file('example.hdml', 1, 18, 17, 1), 'B')),
    pair(file('example.hdml', 2, 3, 26, 12),
      op(file('example.hdml', 2, 3, 26, 5), '&',
        var(file('example.hdml', 2, 3, 26, 1), 'A'),
        var(file('example.hdml', 2, 7, 30, 1), 'B')),
      op(file('example.hdml', 2, 10, 33, 5) '^',
        var(file('example.hdml', 2, 10, 33, 1), 'A'),
        var(file('example.hdml', 2, 14, 37, 1), 'B')))) ]
```

Celem zadania będzie napisanie parsera języka HDML w Prologu, przy użyciu parserów DCG. Dla tego zadania jest również przewidziane zadanie dodatkowe, polegające na napisaniu interpretera.

Zadanie, część 1.
Termin zgłaszania w serwisie SKOS: 17 kwietnia 2017 6:00 AM CEST

Napisz zestaw testów dla parsera języka HDML. Należy posłużyć się następującym szablonem (znajdującym się również w serwisie SKOS):

```
% Definiujemy moduł zawierający testy.
% Należy zmienić nazwę modułu na {imie}_{nazwisko}_tests gdzie za
% {imie} i {nazwisko} należy podstawić odpowiednio swoje imię
% i nazwisko bez wielkich liter oraz znaków diakrytycznych
:- module(imie_nazwisko_tests, [tests/3]).

% Zbiór faktów definiujących testy
% Należy zdefiniować swoje testy
tests(empty_program, input(""), program([])).
tests(invalid, input("def main()"), no).
tests(adder, file('adder.hdml'), yes).
tests(srcpos, input("def main(_) = 1"),
  program([def(main, wildcard(file(test, 1, 10, 9, 1)), num(no, 1))])).
```

rozszerzając definicje predykatu `tests(-Name, -Input, -Ans)`. Oto znaczenia poszczególnych parametrów:

`Name`: atom reprezentujący nazwę testu. Nazwa powinna mówić coś o teście i jednoznacznie go identyfikować.

`Input`: term opisujący dane wejściowe. Powinien mieć jedną z następujących postaci:

- `input(S)` gdzie `S` jest literałem napisowym zawiejącym program do sparsowania;
- `file(Path)` gdzie `Path` jest atomem zawierającym nazwę pliku z programem do sparsowania;

`Ans`: term reprezentujący oczekiwaną odpowiedź. Powinien mieć jedną z następujących postaci:

- `yes` oznacza, że program jest poprawny składniowo;
- `no` oznacza, że program nie jest poprawny składniowo;
- `program(P)` oznacza, że program jest poprawny składniowo oraz reprezentowany jest przez abstrakcyjne drzewo rozbioru `P`. Jeśli decydujesz się umieścić informacje o pozycji w kodzie źródłowym, a parametr `Input` jest postaci `input(S)`, to przyjmij, że ścieżka do pliku to atom `'test'`.

Wymogi formalne
Należy zgłosić pojedynczy plik o nazwie *imię_nazwisko*`_tests.tar.bz2` gdzie za *imię* i *nazwisko* należy podstawić odpowiednio swoje imię i nazwisko bez wielkich liter i znaków diakrytycznych. Nadesłany plik powinien być poprawnym skompresowanym archiwum `tar.bz2` nie zawierającym żadnego katalogu. W archiwum powinny znajdować się **tylko**:

- plik źródłowy napisany w Prologu o nazwie *imię_nazwisko*`_tests.pl` (gdzie *imię* i *nazwisko* są takie jak w nazwie archiwum), który definiuje moduł eksportujący jeden predykat `tests/3` tak jak to opisano w załączonym szablonie.
- wszystkie pliki źródłowe w języku HDML, do których odwołuje się predykat `tests/3` (term `file(Path)` w drugim parametrze). Wszystkie pliki źródłowe w języku HDML powinny mieć rozszerzenie `.hdml`.

Rozwiązania nie spełniające wymogów formalnych nie będą oceniane!

Zadanie, część 2.
Termin zgłaszania w serwisie SKOS: 24 kwietnia 2017 6:00 AM CEST

Napisz moduł eksportujący predykat `parse/3` parsujący programy w języku HDML. Należy posłużyć się następującym szablonem (znajdującym się również w serwisie SKOS):

```
% Definiujemy moduł zawierający rozwiązanie.
% Należy zmienić nazwę modułu na {imie}_{nazwisko} gdzie za
% {imie} i {nazwisko} należy podstawić odpowiednio swoje imię
% i nazwisko bez wielkich liter oraz znaków diakrytycznych
:- module(imie_nazwisko, [parse/3]).

% Główny predykat rozwiązujący zadanie.
% UWAGA: to nie jest jeszcze rozwiązanie; należy zmienić jego
% definicję.
parse(_Path, Codes, Program) :-
  Codes = [], Program = [].
```

zmieniając definicję predykatu `parse(+Path, +Codes, -Program)`. Oto znaczenia poszczególnych parametrów:

`Path`: atom reprezentujący ścieżkę do pliku źródłowego. `Uwaga:` nie wczytuj samemu plików. Parametr ten powinien służyć tylko do poprawnego generowania pozycji w kodzie źródłowym. Jeśli decydujesz się nie generować pozycji w kodzie źródłowym, to po prostu go zignoruj.

`Codes`: Lista kodów poszczególnych znaków składających się na program źródłowy. Możesz założyć, że kodowanie jest zgodne z ASCII dla kodów poniżej 127.

`Program`: Parametr wyjściowy ze sparsowanym programem.

Dla programów niepoprawnych składniowo, Twój predykat powinien albo zawieść, albo za pomocą predykatu `throw/1` zgłosić wyjątek postaci `syntax_error(Reason, Pos)`, gdzie `Pos` jest pozycją w kodzie źródłowym na której pojawił się błąd składniowy, a `Reason` jest dowolnym termem opisującym błąd.

Zgłaszanie błędów i pozycja w kodzie źródłowym
W treści zadania proponujemy, by parser poprawnie generował pozycje poszczególnych elementów w kodzie źródłowym oraz by za pomocą wyjątków zgłaszał błędy składniowe, po to by łatwiej było z nim pracować. Te dwa elementy rozwiązania nie są jednak wymagane. Jeśli nie jesteś pewien jak je poprawnie zaprogramować, to najpierw napisz parser który dla niepoprawnych programów zawodzi, a dla poprawnych programów generuje abstrakcyjne drzewo rozbioru, w którym wszystkie termy reprezentujące pozycje w kodzie źródłowym są równe `no`. Możesz też zapytać się prowadzącego jak w elegancki sposób rozszerzyć swoje rozwiązanie o te dwa nieobowiązkowe elementy.

Wymogi formalne
Należy zgłosić pojedynczy plik o nazwie *imię_nazwisko*`.pl` gdzie za *imię* i *nazwisko* należy podstawić odpowiednio swoje imię i nazwisko bez wielkich liter oraz znaków diakrytycznych. Nadesłany plik powinien być kodem źródłowym napisanym w Prologu, który definiuje moduł eksportujący tylko predykat `parse/3` tak jak to opisano w załączonym szablonie. **Rozwiązania nie spełniające wymogów formalnych nie będą oceniane!**

Uwaga
W serwisie SKOS umieszczono plik `prac3.pl` pozwalający uruchamiać napisane rozwiązanie na przygotowanych testach. Sposób jego uruchamiania znajduje się w komentarzu wewnątrz pliku.