Gentle Compiler Construction System

Gentle - generator kompilatorów oparty na narzędziach lex i yacc (lub GNU flex i bison). Powstał w 1989 roku w GMD Karlsruhe Lab jako Gentle Compiler Construction System. Pozwala on za pomocą jednej deklaratywnej gramatyki (bardzo wygodnej ze względu na składnie podobną do standardowych produkcji), na konstrukcję parsera i kompilatora.
Dzięki zastosowaniom produkcji pozwala na dużą dekompozycję problemu, co powoduje że program jest bardzo prosty w obsłudze, a gramatyki stosunkowo dobrze się czyta (co tłumaczy tak dużą popularność narzędzia w przemyśle[15 lat na rynku]). Poza tym ważnym założeniem przy tworzeniu gentle było zorientowanie na dane, co przekłada się na konstrukcję kompilatora podporządkowaną strukturze danych wejściowych.

Powstały dwie edycje systemu Gentle:

- GENTLE 97 - pierwotna wersja powstała w 1997 r.
- GENTLE 21 - wersja rozszerzona wydana w 2001 r.

Gentle znajduje szerokie zastosowanie w przemyśle informatycznym. Przykładowymi aplikacjami, stworzonymi w Gentle, o których warto wspomnieć są:

- systemy kontrolujące roboty w firmie Volkswagen,
- Security Policy Specification Language[a] tworzone na zlecenie IETF,
- tworzony dla ETSI[b] Testing and Control Notation Language

[a] http://www.ir.bbn.com/projects/pbsm/ipsec-wg-orlando/spsl-sld001.html
[b] http://www.etsi.org/

Najprostszy program dla GENTLE ma postac:

```
'root' print("Hello World!");
```

Element 'root' wskazuje od czego parser ma zacząć analizę. Może to być np. symbol początkowy gramatyki ale również jakaś akcja (jak w tym przypadku predefiniowana akcja print).

Elementy składni

Najważniejsze elementy składni (klasy):

- 'root' - wskazuje na początkowy symbol gramatyki

```
'root' print("Hello World!");
```

- 'nonterm' - nowy nieterminal

- 'token' - terminal
  Jawna deklaracja terminalu, ale nie podaje ona reguły jego tworzenia a tylko informuje że jest taki terminal. Opis jest podawany w innych plikach i dołączany później np. dzięki programowi Reflex. Możemy sami tworzyć pliki z deklaracją tokenu (Nazwa_tokenu.t) Drugim sposobem deklaracji terminala jest po prostu użycie go, np:

```
'nonterm' Instrukcja
    'rule' Instrukcja: "IF" Wyrazenie "THEN" Instrukcja
     "ELSE" Instrukcja
    'rule' Instrukcja: Zmienna ":=" Wyrazenie
```

- 'rule' - reguła (produkcja)
  Składa się ona z pewnego wzorca oraz ciała. Ciało (jeśli jest) stanowi zwykle ciąg akcji i innych reguł.
  Przykładowo załóżmy że zdefiniowaliśmy predykat podający dla danej osoby jej ulubiony kolor:

```
'condition' dajkolor(STRING -> STRING)
    'rule' dajkolor("ala" -> "zielony")
    'rule' dajkolor("ola" -> "niebieski")
```

  Pierwsza linia to "deklaracja" predykatu, określamy w niej zarówno parametry (część przed `->`) jak i rezultat (część po `->`). Następnie widać 2 reguły wraz ze wzorcami używanymi do ich dopasowania, reguły te nie mają żadnego ciała (akcji). Jeśli teraz wywołalibyśmy:

```
dajkolor("ala" -> kolor)
```

  to zmiennej kolor zostanie przypisana wartość "zielony" bo wywołanie pasuje do wzorca w pierwszej regule.

Dodajmy teraz regułę:

```
'rule' dajkolor("jacek" -> kolor): dajkolor("ala" -> kolor)
```

I wywołajmy teraz:

```
dajkolor("jacek" -> jegokolor)
```

Ponieważ to wywołanie pasuje do nowo dodanej reguły, więc zgodnie z nią zostanie najpierw wykonana reguła: `dajkolor("ala" -> kolor)` która zwróci "zielone" i zostanie to podstawione do zmiennej kolor, która następnie zostanie zwrócona z tej reguły i podstawiona do zmiennej jegokolor.

Inny przykład. Załóżmy ze mamy zdefiniowany predykat `ojciec(STRING -> STRING)` podający dla danego człowieka jego ojca. Korzystając z tego możemy teraz zdefiniować predykat dziadek, np w sposób:

```
'condition' dziadek(STRING -> STRING)
    'rule' dziadek(X -> Z): ojciec(X -> Y) ojciec(Y -> Z)
```

- predykaty - zestawy produkcji (reguł)
  Aplikowana jest pierwsza pasująca reguła. predykaty dzielą sie na kilka typów w zależności od reakcji na sytuacje gdy żadna reguła nie pasuje:
  - 'action' - predykat taki powinien z założenia działać zawsze, jeśli brakuje pasującej reguły to jest to uznawane za błąd w specyfikacji i program jest przerywany, przykład:
    ```
    'action' print(T)
    ....
    ```
  - 'condition' - jeśli brakuje pasującej reguły to taki predykat zawodzi, np:
    ```
    'condition' CzyLiczbaPasuje(INT)
                'rule' CzyLiczbaPasuje(5)
    ```
    wywołanie `CzyLiczbaPasuje(2)` zawiedzie ponieważ nie pasuje do tego żadna reguła, ale nie spowoduje błędu wykonania.

Przykład - prosty kalkulator

```
'root' expression(-> X) print(X)

'nonterm' expression(-> INT)
'rule' expression(-> X): expr2(-> X)
'rule' expression(-> X+Y): expression(-> X) "+" expr2(-> Y)
'rule' expression(-> X-Y): expression(-> X) "-" expr2(-> Y)

'nonterm' expr2(-> INT)
'rule' expr2(-> X): expr3(-> X)
'rule' expr2(-> X*Y): expr2(-> X) "*" expr3(-> Y)
'rule' expr2(-> X/Y): expr2(-> X) "/" expr3(-> Y)

'nonterm' expr3(-> INT)
'rule' expr3(-> X): Number(-> X)
'rule' expr3(-> - X): "-" expr3(-> X)
'rule' expr3(-> + X): "+" expr3(-> X)
'rule' expr3(-> X): "(" expression(-> X) ")"

'token' Number(-> INT)
```

- type - definicja nowego typu danych, np:
  ```
  'type' Nazwa_typu
  wartosc1
  wartosc2
  wartosc3
  ..
  ```
  możemy również definiować złożone typy danych(tzw. funktory):
  ```
  'type' Nazwa_typu
  funktor(Typ1,Typ2)
  ```

”Uzbrojeni” w taki mechanizm możemy w prosty sposób zdefiniować listę wartości, np:

```
'type' Postac
Bolek
Lolek
Tola

'type' PrzedszkoleLista
list(Postac,PrzedszkoleLista)
nil
```

Dzięki takiemu mechanizmowi możemy budować takie struktury danych jak: list(Bolek,nil), ale również list(Bolek,list(Lolek,list(Tola,nil))).
Możemy również nadawać nazwy takim elementom struktur:

```
'type' PrzedszkoleLista
list(Postac:Head,PrzedszoleLista:Tail)
nil
```

Typy wbudowane Gentle posiada dwa typy wbudowane, których używa się najczęściej w przetwarzaniu tekstu.

- typ INT - liczby całkowite reprezentowane przez ciąg cyfr [0-9]. Wyrażenia typu INT mogą być budowane z pomocą liczb, podstawowego zbioru operacji(+,-,/,*) oraz nawiasów((,)).
- typ STRING - ciągi znaków ASCI, wyrażenia mogą być zawarte w apostrofach ' lub ". Możliwe jest użycie backslash sequences (`\n \t` etc..)

Abstract syntax

Często pierwszym krokiem przy konstrukcji kompilatorów jest wygenerowane pewnej formy pośredniej programu, z której dopiero jest generowany wynikowy kod.
Weźmy dla przykładu proste wyrażenia arytmetyczne. Najpierw musimy wprowadzić nowy typ składający się z możliwych funktorów:

```
'type' Expr
    plus(Expr,Expr)
    minus(Expr,Expr)
    mult(Expr,Expr)
    neg(Expr)
    num(INT)
```

Następnie tworzymy reguły, które konstruują odpowiednie funktory z danego wyrażenia, np: `2+3*4` chcemy mieć zamienione na `plus(num(2), mult(num(3), num(4)))`

```
'nonterm' expression(-> Expr)
    'rule' expression(-> X): expr2(-> X)
    'rule' expression(-> plus(X,Y)): expression(-> X) "+" expr2(->
    'rule' expression(-> minus(X,Y)): expression(-> X) "-" expr2(-
'nonterm' expr2 (-> Expr)
    'rule' expr2(-> X): expr3(-> X)
    'rule' expr2(-> mult(X,Y)): expr2(-> X) "*" expr3(-> Y)
'nonterm' expr3 (-> Expr)
    'rule' expr3(-> num(X)): Number(-> X)
    'rule' expr3(-> neg(X)): "-" expr3(-> X)
    'rule' expr3(-> X): "+" expr3(-> X)
    'rule' expr3(-> X): "(" expression(-> X) ")"
'token' Number (-> INT)
```

Jest to typowa jednoznaczna gramatyka wyrażen arytmetycznych zapisana w GENTLE.

Następnie musimy utworzyć drugi zestaw reguł, które będą przechodziły przez utworzone już drzewo funktorów i np. obliczały wartość wyrażenia:

```
'action' eval(Expr -> INT)
    'rule' eval(plus(X1,X2) -> N1+N2): eval(X1 -> N1) eval(X2 -> N
    'rule' eval(minus(X1,X2) -> N1-N2): eval(X1 -> N1) eval(X2 ->
    'rule' eval(mult(X1,X2) -> N1*N2): eval(X1 -> N1) eval (X2 ->
    'rule' eval(neg(X) -> -N): eval(X -> N)
    'rule' eval(num(N) -> N)
```

Użycie całego tego mechanizmu może wyglądać następująco:

```
'root' expression(-> X) eval(X -> N) print(N)
```

Najpierw do zmiennej X generowane jest drzewo funktorów (expression). Następnie przechodzimy po tym drzewie i obliczamy wynik (eval), który na końcu wypisujemy (print).

Poza tym proste okazuje się rozszerzanie tak zdefiniowanej składni. Na przykład chcąc stworzyć program różniczkujący, jedyne co musimy dodać do już istniejącej składni to:

```
'rule' expr3(-> x): "x"

'action' deriv (Expr -> Expr)

'rule' deriv(mult (U,V) -> plus(mult(Ud,V), mult(U,Vd))):
deriv(U -> Ud)
deriv(V -> Vd)
'rule' deriv(div(U,V) -> div(minus(mult(Ud,V),mult(U,Vd)),mult(V,V
deriv(U -> Ud)
deriv(V -> Vd)
'rule' deriv(plus(U,V) -> plus(Ud, Vd)):
deriv(U -> Ud)
deriv(V -> Vd)
'rule' deriv(minus(U,V) -> minus(Ud, Vd)):
deriv(U -> Ud)
deriv(V -> Vd)
'rule' deriv(num(N) -> num(0))
'rule' deriv(x -> num(1))
```

a następnie podmienić akcję początkową:

```
'root' expression(->X) deriv(X->D) print(D)
```

Zmienne globalne

Mimo ze mechanizmy GENTLE jako języka deklaratywnego są dość silne, to jednak czasem stajemy przed koniecznością użycia zmiennych. Przykładem może być język w którym instrukcja wyjścia, może znajdować się tylko w obrębie danego bloku (np. break w pętlach w C). Tworzenie nowych produkcji dla każdej z takich sytuacji może być kłopotliwe, dlatego lepiej jest użyć flagi oznaczającej czy w danym bloku się znajdujemy. Zmienną deklarujemy instrukcją:

```
'var' Nazwa: Typ
```

np instrukcja:

```
'var' CurLoopContext: LoopContext
```

deklarująca zmienna CurLoopContext typu LoopContext. Zmiennych możemy używać tylko i wyłącznie w regułach. Wartość zmiennej odczytujemy konstrukcją:

```
CurLoopContext -> A
```

Zmiany wartości dokonujemy instrukcją:

```
CurLoopContext <- inside
```

Przykładowo mając już zdefiniowaną gramatykę dla pętli i Body w języku C:

```
'rule' ProcessStatement(loop(Body)):
CurLoopContext -> OldLoopContext
CurLoopContext <- inside
ProcessStatement(Body)
CurLoopContext <- OldLoopContext

'rule' ProcessStatement(break):
CurLoopContext -> K
CheckContextForExit(K)
```

w bardziej skomplikowanych przypadkach możliwe jest użycie struktury konstrukcją:
```
'table' Nazwa ( Pole1:Typ1, Pole2:Typ2,... )
```

Do poszczególnych pól możemy się odwołać instrukcją:
```
NazwaZmiennejStruktualnej'NazwaPola
```

Wybór produkcji w oparciu o koszt

Czasem zdarza się, że chcemy by kompilator zdecydował którą produkcję wybrać na podstawie dostarczonych kosztów. Np chcemy przetłumaczyć if(! expr), na perla. Mamy dwie możliwości: if(!expr) i unless(expr) (oczywiście dążymy do wyboru tej drugiej opcji). Możemy wymusić odpowiednie zachowanie kompilatora przez zastosowanie konstrukcji:

```
'choice' ifnot(Expr)

'rule' ifnot(if(not(Expr))):
expression(Expr)
IFNOT(Expr)
$ 20

'rule' ifnot(if(not(Expr))):
expression(Expr)
UNLESS(EXPR)
$ 10
```

instrukcje te spowodują, że koszt(liczba po dolarze na końcu produkcji) parsowania każdego z wyrażeń będzie wyliczany dla każdej możliwości i wybierana będzie opcja optymalna (o najniższym koszcie sumarycznym).

Użycie w dużych projektach

Jak każde poważne narzędzie programistyczne także GENTLE posiada konstrukcje pozwalające na użycie go w dużych projektach:

- Module - predykat wyszczególniający przestrzeń nazw w jakiej znajduje się dana część projektu, np:
  `'Module' CLoops;`

- Use - definiuje listę modułów z jakich korzystamy przy konstrukcji danej części kodu, np:
  `'Use' CLoops, CExpressions, CStatments`

- Export - lista identyfikatorów które są widoczne dla innych modułów włączających naszą przestrzeń nazw, np:
  `'Export' forLoop, whileLoop`

W przypadku użycia modułów ważne jest aby tylko jeden z nich zawierał predykat 'root'.

Inne nieopisane w referacie własności GENTLE:

- Joker - pasująca do wszystkiego część produkcji
- Nazwane dopasowania w sytuacjach kiedy chcemy widzieć całe dopasowanie, a nie tylko jego części
- Predykat where
- Struktury sterujące alternatywa i instrukcja warunkowa
- Predykaty zdefiniowane do porównywania wartości (eq,ne,lt,le..)

Duży przykład: www.ds5.agh.edu.pl/~kubek2k/gentle/my

http://gentle.compilertools.net
http://www.ds5.agh.edu.pl/~kubek2k/gentle
http://www.google.com

KONIEC