GENTLE

http://student.agh.edu.pl/~wstanisz/gentle

- gentle.pdf [prezentacja]
- gentle-97.tar.gz [źródła]

Po rozpakowaniu gentle-97.tar.gz wejdź kolejno do katalogów: ***gentle, lib*** `i` ***reflex***
i w każdym wpisz ***./build***

W katalogu ***examples*** znajdują się programy omawiane podczas prezentacji.

GENTLE

Narzędzie do konstruowania kompilatorów

GENTLE - Co będzie?

- Co to jest GENTLE
- Gdzie jest używany
- Co potrafi
- Opis narzędzia
- Przykład

GENTLE - Wprowadzenie

Jest uniwersalnym narzędziem dla projektantów kompilatorów i implementatorów języków

- Używany w przemyśle i badaniach naukowych
- Posiada rozmaite możliwości począwszy od analizy przez transformację do syntezy
- Jest darmowy

GENTLE - Historia i edycje

- Gentle powstał w *Niemiecim Narodowym Centrum Rozwoju Technologii Informatycznych* w 1989 r.
- Stale się rozwija.

- Gentle jest dostępny w 2 dystrybucjach:
  - GENTLE 97
  - GENTLE 21

GENTLE - Wykorzystanie 1/2

- *Nixdorf Computers* dla implementacji *Combined Object-Oriented Language*
- W przemyśle motoryzacyjnym w oprogramowaniu do sterowania robotów np. *Volkswagen.*
- Do stworzenia kompilatora dla *Security Policy Specification Language*
- *Siemens* w aplikacjach telekomunikacyjnych- kompilator dla *Testing and Test Control Notation*
- *Ritlabs* twórcy *The Bat* 2005 r.

GENTLE - Wykorzystanie 2/2

Typowe wykorzystanie:

- Zorientowany obiektowo język symulacji wysokiego poziomu
- Język programowania dla sieci neuronowych
- Translator z Mathematica do C
- Język programowania dla robotów przemysłowych
- Zorientowany obiektowo język dla systemów informacyjnych
- Język dla programowania logicznego i funkcjonalnego
- Język zapytań dla OODBMS

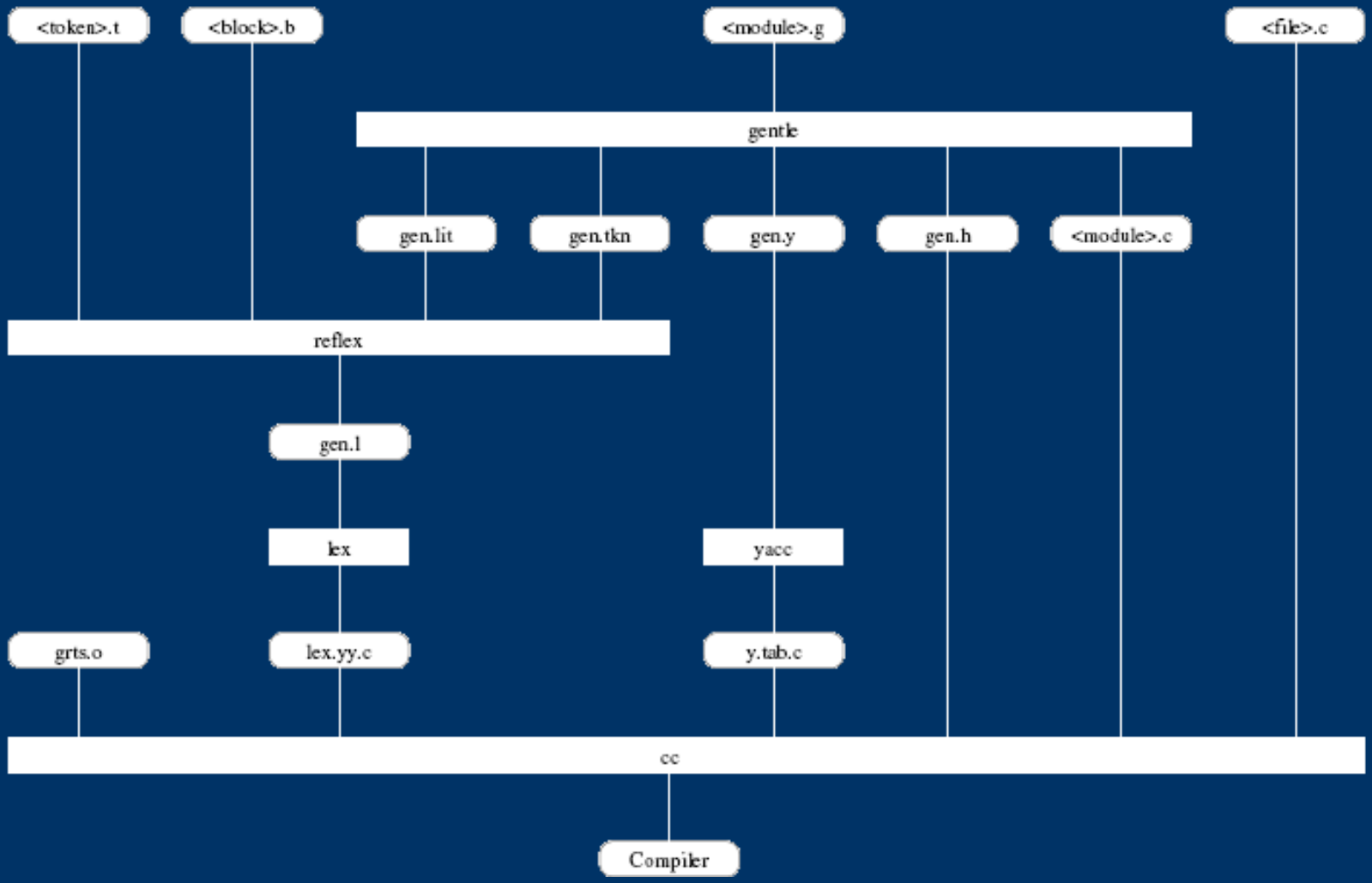
GENTLE - Możliwości

- Rozpoznawanie języka
- Definiowanie abstrakcyjnych drzew składniowych
- Dopasowywanie wzorców
- Smart Traversal
- Prosta translacja *source-to-source*
- Optymalny wybór kodu dla mikroprocesorów

GENTLE - Udogodnienia

- Udostępnia zunifikowane środowisko pracy i jednolitą notację dla wszystkich zadań.
- Bazuje na rekurencyjnych definicjach i indukcji strukturalnej.
- Zapewnia niezależny opis różnych konstrukcji- co umożliwia zrozumienie całego systemu poprzez analizę jego poszczególnych części.
- *Data-oriented methodology*: struktura danych jest „odbijana” w strukturze algorytmu.

GENTLE - Ogólny shemat



GENTLE - Postawy 1/4

- *Gentle* pozwala użytkownikowi na definiowanie rekurencyjnych typów, poprzez wypisanie alternatywnych konstrukcji.

```
Expr =
        plus(Expr,Expr),
        minus(Expr,Expr),
        const(INT)
```

GENTLE - Postawy 2/4

- Programy w *Gentle* wyrażone są w postaci reguł:

```
G : A B C
```

Te reguły moga być interpretowane w różny sposób:

- jako reguły gramatyki
- jako wyrażenie logiczne
- albo też w proceduralny sposób

GENTLE - Postawy 3/4

- Parametry mogą być w regułach:
  („->” oddziela wejście od wyjścia)

```
AddingExpression(-> plus(X1, X2)):
      AddingExpression(-> X1) "+" Primary(-> X2)
```

- a także w schematach transformacji:

```
Eval(plus (X1, X2) -> N1+N2):
    Eval(X1 -> N1)
    Eval(X2 -> N2).
Eval(minus(X1, X2) -> N1-N2):
    Eval(X1 -> N1)
    Eval(X2 -> N2).
Eval(const(N) -> N).
```

GENTLE - Postawy 4/4

- *Unparsing* może być wyrażony analogicznie:

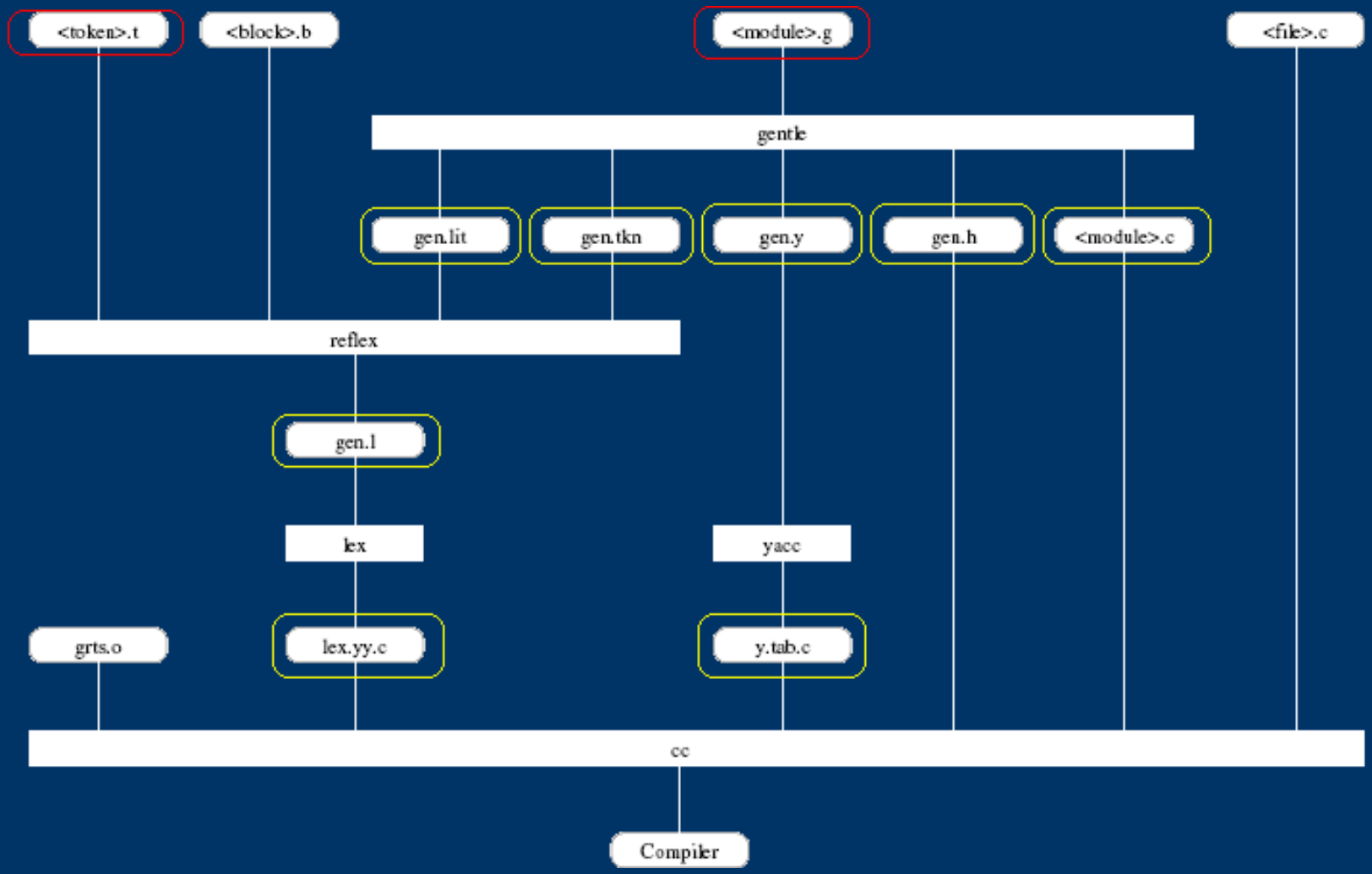
```
Code(plus(X1, X2) -> Reg2) :
        Code(X1 -> Reg1)
        Code(X2 -> Reg2)
        Emit("add", Reg1, Reg2)
```

GENTLE - Przykład

1. Parser wyrażeń arytmetycznych
2. Kalkulator
3. Definiowanie abstrakcyjnej składni
4. Tłumaczenie ze zwykłej do abstrakcyjnej składni

   Przykłady wykorzystania abstrakcyjnej składni:
5. Kalkulator
6. Pochodna wyrażenia
7. Generacja kodu

GENTLE - Przykład



1. Parser 1/3

```
'root' expression
'nonterm' expression
   'rule' expression: expr2
   'rule' expression: expression "+" expr2
   'rule' expression: expression "-" expr2
'nonterm' expr2
   'rule' expr2: expr3
   'rule' expr2: expr2 "*" expr3
   'rule' expr2: expr2 "/" expr3
'nonterm' expr3
   'rule' expr3: Number(-> N)
   'rule' expr3: "-" expr3
   'rule' expr3: "+" expr3
   'rule' expr3: "(" expression ")"
'token' Number(-> INT)
```

1. Parser 2/3

- Parser zapisany jest jako gramatyka.
- Nieterminale:

```
'nonterm' expression
```

- są zbudowane z fraz- reprezentowanych przez reguły:

```
'rule' expression: expression "+" expr2
```

1. Parser 3/3

- Symbole terminalne pojawiają się w cudzysłowie np: ”+” albo są wprowadzane w deklaracjach tokenów (ale nie specyfikowane).

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

- Symbol początkowy gramatyki:

```
'root' expression
```

2. Kalkulator 1/3

```
'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)
```

2. Kalkulator 2/3

- Gramatykę przekształciliśmy w kalkulator, który na wejściu dostaje wyrażenie arytmetyczne i wypisuje jego wartość.
- Każdy nieterminal dostał parametr specyfikujący jego wartość.

```
'nonterm' expression(-> INT)
```

- Po ewaluacji `expression` wypisywana jest wartosć.

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

2. Kalkulator 3/3

- Reguła:

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

  specyfikuje jak obliczana jest wartość `expression` z wartości jej składowych.

- np: `1+2*3` warotść `expression(->X)` określa `X`(czyli `1`), analogicznie wartość `expr2(->Y)` określa `Y`(czyli `6`), więc wartość `expression(->X+Y)` wynosi suma: `X+Y`,(czyli `7`).

3. Abstrakcyjna składnia 1/2

- Abstrakcyjna składnia jest zwykle lepiej dopasowana do rożnych przekształceń niż pierwotna specyfikacja

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

3. Abstrakcyjna składnia 2/2

- Wartości tego typu specyfikowane poprzez alternatywy. Np:

```
num(INT)
```

oznacza, że funktor `num` z argumentem `INT` reprezentuje wartość `Expr` (np. `num(15)`)

- albo:

```
mult(Expr, Expr)
```

oznacza, że funkor `mult` może być użyty z dwoma argumentami typu `Expr` np. `num(14)` i `num(15)`.

Co oznacza że term `mult(num(14, num(15))` jest także wartoscią typu `Expr`.

4. Translacja 1/2

```
'root' expression(-> X) print(X)
'nonterm' expression(-> Expr)
   'rule' expression(-> X): expr2(-> X)
   'rule' expression(->plus(X,Y)):expression(->X) "+" expr2(->Y)
   'rule' expression(->minus(X,Y)):expression(->X) "-" expr2(->Y)
'nonterm' expr2(-> Expr)
   'rule' expr2(-> X): expr3(-> X)
   'rule' expr2(-> mult(X,Y)): expr2(-> X) "*" expr3(-> Y)
   'rule' expr2(-> div(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)
```

4. Translacja 2/2

- Translacja z konkretnej składni do abstrakcyjnej wygląda analogicznie jak w przypadku kalkulatora z tą jednak różnicą, że zamiast wartości liczbowej mamy ciąg znaków odpowiadający danemu wyrażeniu. Np:

```
'rule' expr3(-> neg(X)): "-" expr3(-> X)
```

  odpowiada:

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

- Podobnie `'root' expression(-> X) print(X)` powoduje wypisanie przetłumaczonego wyrażenia

5. Kalkulator 1/2

Przetwarzanie składni abstrakcyjnej

na przykładzie kalkulatora

- Postać pierwszej produkcji jest nastepująca:

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

- Gdzie wartością **`expression`** jest łańcuch terminali **`X`** o typie **`Expr`**, a **`eval`** to procedura która dostaje na wejściu parametr typu **`Expr`** i zwraca parametr typu **`INT`**.

5. Kalkulator 2/2

- Mapowanie definiują poniższe reguły:

```
'action' eval (Expr -> INT)

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

6. Pochodna wyrażenia 1/2

Przetwarzanie składni abstrakcyjnej

różniczkowanie wyrażenia

- Wzbogacimy o „x” wyrażenia, które będziemy akceptować. I będziemy różniczkować wyrażenie ze względu na „x”.

- Wystarczy dodać do konkretnej składni regułę:
  `'rule' expr3(-> x): "x"`
- Do abstrakcyjnej `x`, oraz dopisać....

6. Pochodna wyrażenia 2/2

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

'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))
```

7. Generacja kodu 1/2

- Parser kompilatora przekształca konkretny program do składni abstrakcyjnej, natomiast generator kodu przekształca składnię abstrakcyjną w konkretny program. Takie mapowanie może być wyspecyfikowane podobnie jak w poprzednich przykładach z tą różnicą, że wyrażenia składni abstrakcyjnej stają się parametrami sterującymi wyborem reguł.

7. Generacja kodu 2/2

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

'action' code (Expr)
'rule' code(plus(X1, X2)):code(X1) code(X2) print("plus")
'rule' code(minus(X1, X2)):code(X1) code(X2) print("minus")
'rule' code(mult(X1, X2)):code(X1) code(X2) print("mult")
'rule' code(div(X1, X2)):code(X1) code(X2) print("div")
'rule' code(neg(X)): code(X) print("neg")
'rule' code(num(N)): print(N)
```

GENTLE - Podsumowanie

- Wysoki poziom abstrakcji uwalnia od szczegółów implementacyjnych
- Metodologia wg której zbudowany jest *Gentle* w prosty sposób sprzyja tworzeniu kompilatorów
- Jednolite środowisko pracy czyni język prostym i łatwym do nauczenia się

DZIĘKUJEMY ZA UWAGĘ