Metody Kompilacji
Wykład 13

Prosty Translator

Translator dla prostych wyrażeń

Schemat translacji sterowanej składnią często służy za specyfikację translatora.

Schemat na następnym slajdzie zostanie użyty jako definicja translacji, która przekształca wyrażenia arytmetyczne na odwrotną notację polską.

Prosty Translator

Translator dla prostych wyrażeń

expr -> *expr* + *term* { print('+') }

| *expr* - *term* { print('-') }

| *term*

term -> 0 { print ('0') }

| 1 { print(‘1’) }

. . .

| 9 { print ('9') }

Gramatyka jest lewostronnie rekurencyjna

Prosty Translator

Zastosowanie analizy zstępującej

Prosty Translator

Translator dla prostych wyrażeń

Gramatyka jest lewostronnie rekurencyjna, więc ”parser przewidujący” jej nie obsługuje.

Mamy konflikt: z jednej strony potrzebujemy gramatyki, która ułatwia translację, z drugiej strony musimy mieć zupełnie inną gramatykę, która ułatwia parsowanie.

Prosty Translator

Translator dla prostych wyrażeń

- Rozwiązanie polega na tym, aby rozpocząć od gramatyki dla łatwej translacji i starannie przekształcić ją w celu ułatwienia parsowania.
- Eliminując rekurencję lewostronną, możemy uzyskać gramatykę odpowiednią do zastosowania parsera przewidującego.

Prosty Translator

Składnia abstrakcyjna i konkretna

- W abstrakcyjnym drzewie syntaktycznym dla wyrażenia, każdemu węzłowi wewnętrznemu odpowiada operator;

  dzieci węzła reprezentują argumenty operatora.

Prosty Translator

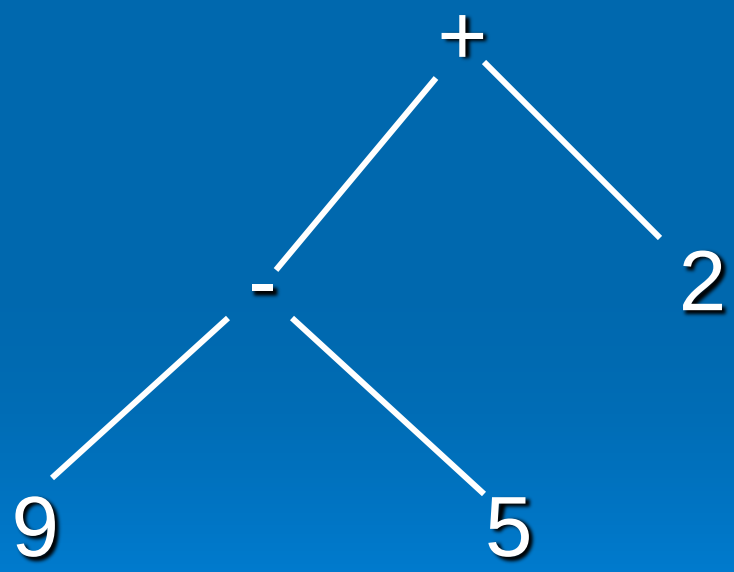
Składnia abstrakcyjna i konkretna

- Bardziej ogólnie, każda konstrukcja języka programowania może być obsługiwana przez operator dla tej konstrukcji, operandami tego operatora są semantycznie sensowne elementy tej konstrukcji.

Prosty Translator

Składnia abstrakcyjna i konkretna

- Drzewo abstrakcyjne dla 9-5+2:



Prosty Translator

Składnia abstrakcyjna i konkretna

- Abstrakcyjne drzewo syntaktyczne, lub po prostu drzewo syntaktyczne, przypomina w niektórym stopniu drzewo parsowania.
- Jednak w drzewie syntaktycznym, węzły wewnętrzne reprezentują konstrukcje programistyczne, podczas gdy w drzewie parsowania, węzły wewnętrzne reprezentują nieterminale.

Prosty Translator

Składnia abstrakcyjna i konkretna

- Wiele symboli nieterminalnych gramatyki reprezentują konstrukcje programistyczne, ale inne to są tak zwane "pomocnicy", jak na przykład symbole *terms* i *factors* (wprowadzone na wykładzie 2) stosowane są do wyprowadzenia wyrażeń arytmetycznych.

Prosty Translator

Składnia abstrakcyjna i konkretna

- W drzewie syntaktycznym, te „pomocnicy” zazwyczaj nie są potrzebne i są usuwane.
- Aby podkreślić kontrast, drzewo parsowania jest czasami nazywane konkretnym drzewem syntaktycznym, natomiast odpowiednia gramatyka jest nazywana konkretną składnią języka.

Prosty Translator

Składnia abstrakcyjna i konkretna

- Wskazane jest, aby schemat translacji był oparty na gramatyce, której drzewa parsowania byłyby tak blisko drzew syntaktycznych, jak to tylko możliwe.

Prosty Translator

Dostosowanie schematu translacji

- Technika eliminacji rekurencji lewostronnej (została przedstawiona na wykładzie nr 4) może być zastosowana do produkcji zawierających akcje semantyczne.

Prosty Translator

Dostosowanie schematu translacji

Rozważmy gramatykę:

expr -> *expr* + *term* { print('+') }

| *expr* - *term* { print('-') }

| *term*

term -> 0 { print ('0') }

| 1 { print(‘1’) }

. . .

| 9 { print ('9') }

Prosty Translator

Dostosowanie schematu translacji

- Zgodnie z techniką eliminacji rekurencji lewostronnej, w naszym przykładzie *A* jest to *expr* i są dwie produkcje lewostronnie rekurencyjne dla *expr* oraz jedna, która nie jest rekurencyjna.

Prosty Translator

Dostosowanie schematu translacji

- Technika dokonuje transformacji produkcji

  $A \rightarrow A\alpha | A\beta \mid \gamma$

na produkcje

$A \rightarrow \gamma R$

$R \rightarrow \alpha R \mid \beta R \mid \varepsilon$

Prosty Translator

Dostosowanie schematu translacji

- Musimy także przekształcić produkcje z akcjami semantycznymi.

- Akcje semantyczne po prostu są traktowane, jak gdyby były one terminalami.

Prosty Translator

Dostosowanie schematu translacji

Niech:

A = expr

α = + term { print ('+') }

β = - term { print('-') }

γ = term

Prosty Translator

Dostosowanie schematu translacji

Wtedy transformacja usuwania rekurencji lewostronnej produkuje następujący schemat:

Usuwanie rekurencji lewostronnej

$expr \rightarrow term \;\; rest$

$rest \rightarrow + \; term$ { print('+') } $rest$

$\quad\quad\quad\quad | \; - \; term$ { print('-') } $rest$

$\quad\quad\quad\quad | \; \varepsilon$

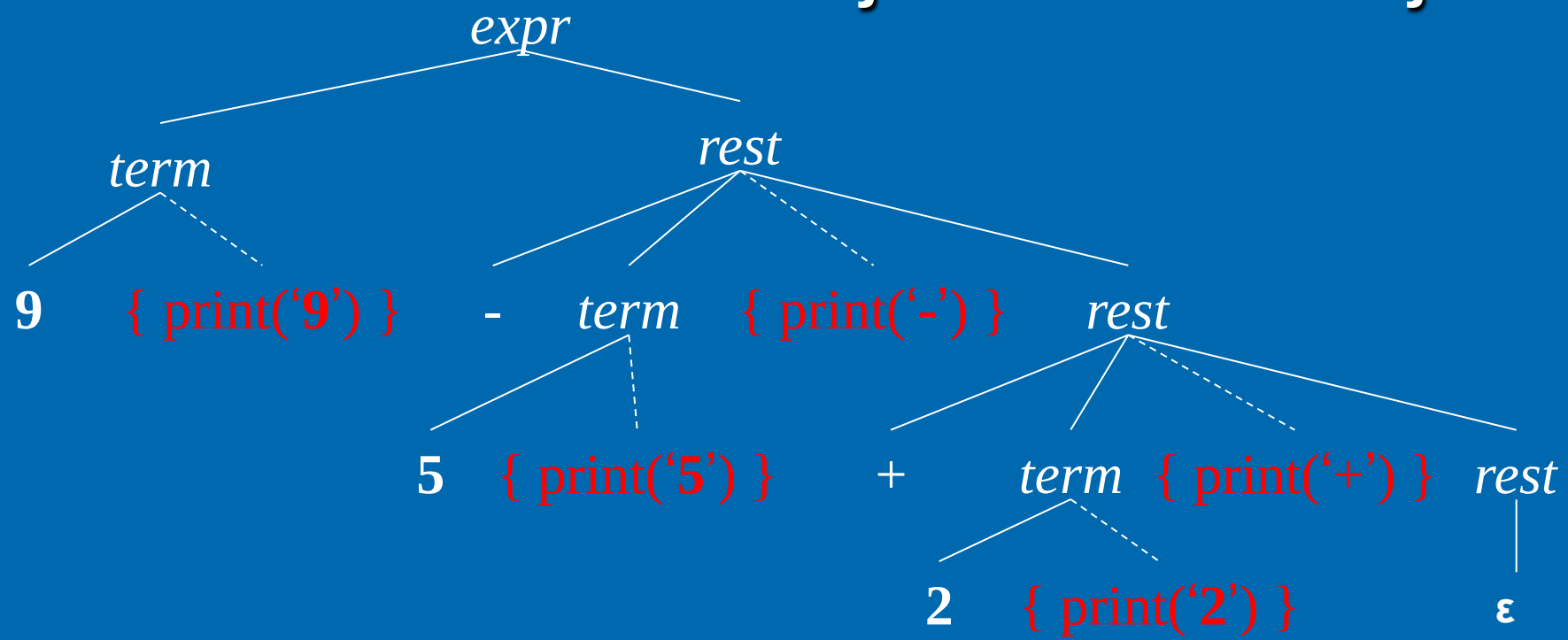
$term \rightarrow \mathbf{0}$ { print('0') }

$term \rightarrow \mathbf{1}$ { print('1') }

$\quad \ldots$

$term \rightarrow \mathbf{9}$ { print('9') }

Usuwanie rekurencji lewostronnej



$expr \rightarrow term\ rest$

$rest \rightarrow + term$ { print('+') } $rest$

$\quad | \quad - term$ { print('-') } $rest$

$\quad | \quad \varepsilon$

$term \rightarrow$ **0** { print('0') } | **1** { print('1') } | … | **9** { print('9') }

Pseudokod dla nieterminali

$expr \rightarrow term \ \ rest$

$rest \rightarrow + \ term \quad \{ \text{print}(\text{'+'}) \} \quad rest$

$\mid - \ term \quad \{ \text{print}(\text{'-'}) \} \quad rest$

$\mid \varepsilon$

$term \rightarrow \textbf{dig} \quad \{ \text{print}(\text{'dig'}) \}$

```
void expr() {
    term(); rest();
}

void rest() {
    if ( lookahead == '+' )  {
       match('+'); term();
       print('+'); rest();
    }
    else if ( lookahead == '-' ) {
       match('-'); term();
       print('-'); rest();
    }
    else { }  //do nothing with the input
}

void term() {
    if ( lookahead is a digit )  {
        t = lookahead; match(lookahead);
        print(t);
    }
    else
        report("syntax error");
}
```

Pseudokod dla nieterminali

$expr \rightarrow term \ \ rest$

$rest \rightarrow + \ term \quad \{ \text{print}(\text{'+'}) \} \quad rest$

$\mid - \ term \quad \{ \text{print}(\text{'-'}) \} \quad rest$

$\mid \varepsilon$

$term \rightarrow \textbf{dig} \quad \{ \text{print}(\text{'dig'}) \}$

```
void expr() {
    term(); rest();
}

void r
    if
    }
    else if ( lookahead ==      ) {
        match('-'); term();
        print('-'); rest();
    }
    else { }  //do nothing with the input
}

void term() {
    if ( lookahead is a digit )  {
        t = lookahead; match(lookahead);
        print(t);
    }
    else
        report("syntax error");
}
```

Funkcja expr implementuje produkcję dla nieterminala *expr.*

Pseudokod dla nieterminali

Funkcja rest implementuje trzy produkcje dla nieterminala *rest*. Ona stosuje pierwszą produkcję, jeśli symbolem bieżącym jest znak plus, drugą produkcję jeśli symbolem bieżącym jest znak minus, i produkcję *rest* ->ε we wszystkich innych przypadkach.

```
void expr() {
    term(); rest();
}

void rest() {
    if ( lookahead == '+' )  {
        match('+'); term();
        print('+'); rest();
    }
    else if ( lookahead == '-' ) {
        match('-'); term();
        print('-'); rest();
    }
    else { }  //do nothing with the input
}

void term() {
    if ( lookahead is a digit )  {
         t = lookahead; match(lookahead);
         print(t);
    }
    else
        report("syntax error");
}
```

Pseudokod dla nieterminali

expr → *term rest*

rest → + *term* { print(‘+’) } *rest*

| – *term* { print(‘-’) } *rest*

| ε

term → **dig** { print(‘dig’) }

```
void expr() {
                                        {
                                    the input
}
void term() {
    if ( lookahead is a digit )  {
        t = lookahead; match(lookahead);
        print(t);
    }
    else
        report("syntax error");
}
```

Dziesięć produkcji dla *term* generują dziesięć cyfr.
Ponieważ każda z tych produkcji generuje i drukuje cyfrę, ten sam kod realizuje je wszystkie.

Pseudokod dla nieterminali

$expr \rightarrow term \ \ rest$

$rest \rightarrow + \ term \quad \{ \text{print}(\text{'+'}) \} \quad rest$

$\mid - \ term \quad \{ \text{print}(\text{'-'}) \} \quad rest$

$\mid \varepsilon$

$term \rightarrow \textbf{dig} \quad \{ \text{print}(\text{'dig'}) \}$

```
void expr() {
    term(); rest();
}

void rest() {
                                        {
                                        the input
}

void term() {
    if ( lookahead is a digit )  {
        t = lookahead; match(lookahead);
        print(t);
    }
    else
        report("syntax error");
}
```

Jeśli test zakończy się pomyślnie, zmienna t przechowuje cyfrę reprezentowaną przez symbol bieżący.

Pseudokod dla nieterminali

expr → *term rest*

```
void expr() {
    term(); rest();
}
```

rest → + *term* { print('+') } *rest*

| – *term* { print('-') }

| ε

```
void rest() {
    if ( lookahead == '+' ) {
        match('+'); term();
```

```
{
```

```
the input
```

Należy pamiętać, że funkcja match zmienia symbol bieżący, więc cyfra musi być zapisana w celu późniejszego wydrukowania.

term → **dig** { print('dig') }

```
void term() {
    if ( lookahead is a digit ) {
        t = lookahead; match(lookahead);
        print(t);
    }
    else
        report("syntax error");
}
```

Pseudokod

$expr \rightarrow term \ \ rest$

$rest \rightarrow + \ term \quad \{ \text{print}(\text{'+'}) \} \quad rest$

$\mid - \ term \quad \{ \text{print}(\text{'-'}) \} \quad rest$

$\mid \varepsilon$

$term \rightarrow \textbf{dig} \quad \{ \text{print}(\text{'dig'}) \}$

Gdy podczas wykonywania procedury, ostatnią czynnością jest wywołanie rekurencyjne tej samej procedury, to mówimy, że jest to rekurencja ogonowa (*tail recursive*).

```
void

}

void rest() {
    if ( lookahead == '+' ) {
        match('+'); term();
        print('+'); rest();
    }
    else if ( lookahead == '-' ) {
        match('-'); term();
        print('-'); rest();
    }
    else { }  //do nothing with the input
}

void term() {
    if ( lookahead is a digit ) {
        t = lookahead; match(lookahead);
        print(t);
    }
    else
        report("syntax error");
}
```

Pseudokod

$expr \rightarrow term \;\; rest$

$rest \rightarrow + \; term \quad \{ \text{print}(\text{'+'}) \} \;\; rest$

$\mid - \; term \quad \{ \text{print}(\text{'-'}) \} \;\; rest$

$\mid \varepsilon$

$term \rightarrow \textbf{dig} \quad \{ \text{print}(\text{'dig'}) \}$

Wywołania rekurencyjne mogą być zastąpione przez iteracje.

```
void

}

void rest() {
    if ( lookahead == '+' ) {
        match('+'); term();
        print('+'); rest();
    }
    else if ( lookahead == '-' ) {
        match('-'); term();
        print('-'); rest();
    }
    else { }  //do nothing with the input
}

void term() {
    if ( lookahead is a digit ) {
        t = lookahead; match(lookahead);
        print(t);
    }
    else
        report("syntax error");
}
```

Pseudoko

Dla procedury bez parametrów, rekurencję ogonową można wymienić na instrukcję skoku na początek procedury.

expr → *term rest*

rest → + *term* { print('+') } *rest*

| – *term* { print('-') } *rest*

| ε

term → **dig** { print('dig') }

```
v
}

void rest() {
    if ( lookahead == '+' ) {
        match('+'); term();
        print('+'); rest();
    }
    else if ( lookahead == '-' ) {
        match('-'); term();
        print('-'); rest();
    }
    else { }  //do nothing with the input
}

void term() {
    if ( lookahead is a digit ) {
        t = lookahead; match(lookahead);
        print(t);
    }
    else
        report("syntax error");
}
```

Uproszczenie kodu

Dopóki symbolem bieżącym jest znak plus lub minus, procedura rest wywołuje procedurę term. W przeciwnym razie, wykonywanie pętli while się kończy.

```
void rest() {
    if ( lookahead == '+' )  {
       match('+'); term();
       print('+'); rest();
    }
    else if (lookahead == '-') {
       match('-'); term();
       print('-'); rest();
    }
    else { }
```

```
void  e() {
  while ( true ) {
    if ( lookahead == '+' ) {
      match('+'); term();
      print('+'); continue;
    }
    else if (lookahead == '-') {
       match('-'); term();
       print('-'); continue;
    }
    break;
  }
}
```

Uproszczenie kodu

Gdy rekurencja ogonowa jest realizowana przez iteracje, zostaje tylko jedno wywołanie procedury rest z wewnątrz procedury expr. Dwie procedury mogą być zatem zintegrowane w jedną przez zastąpienie procedury rest() przez jej ciało.

```
void rest() {
    if ( lookahead == '+' )  {
       match('+'); term();
       print('+'); rest();
    }
    else if (lookahead == '-') {
       match('-'); term();
       print('-'); rest();
    }
    else { }
```

```
void [illegible]() {
  while ( true ) {
    if ( lookahead == '+' ) {
      match('+'); term();
      print('+'); continue;
    }
    else if (lookahead == '-') {
       match('-'); term();
       print('-'); continue;
    }
    break;
  }
}
```

Klasa Parser w Javie

```
import java.io.*;

class Parser {
  static int lookahead;

  public Parser() throws IOException {
    lookahead = System.in.read();
  }

  void expr() {
    term();
    while ( true ) {
    if ( lookahead == '+' ) {
      match('+'); term();
      System.out.write('+');
      continue;
    }
    else if (lookahead == '-') {
       match('-'); term();
       System.out.write('-');
       continue;
    }
    else return;
  }
```

```
void term() throws IOException {
  if (Character.isDigit((char)lookahead){
     System.out.write((char)lookahead);
     match(lookahead);
  }
  else throw new Error("syntax error");
}

void match(int t) throws IOException {
  if ( lookahead == t )
     lookahead = System.in.read();
  else throw new Error("syntax error");
 }
}
```

Klasa Parser w Javie

```
import java.io.*;

class Parser {
  static int lookahead;

  public Parser() throws IOException {
    lookahead = System.in.read();
  }

  void expr() {
    term();
    while ( true ) {
    if ( lookahead == '+' ) {
      match('+'); term();
      System.out.write('+');
      continue;
    }
    else if (lookahead == '-') {
       match('-'); term();
       System.out.write('-');
       continue;
    }
    else return;
  }
```

Klasa Parser zawiera zmienną `lookahead` i funkcje Parser, expr, term, and match.

```
void term() throws IOException {
  if (Character.isDigit((char)lookahead){
     System.out.write((char)lookahead);
     match(lookahead);
  }
  else throw new Error("syntax error");
}

void match(int t) throws IOException {
  if ( lookahead == t )
     lookahead = System.in.read();
  else throw new Error("syntax error");
 }
}
```

Klasa Parser w Javie

```
import java.io.*;

class Parser {
  static int lookahead;

  public Parser() throws IOException {
    lookahead = System.in.read();
  }

  void expr() {
    term();
    while ( true ) {
    if ( lookahead == '+' ) {
      match('+'); term();
      System.out.write('+');
      continue;
    }
    else if (lookahead == '-') {
       match('-'); term();
       System.out.write('-');
       continue;
    }
    else return;
  }
```

```
void te
  if (C
     Sy
     ma
  }
  else th
}

void match(int t) throws IOException {
  if ( lookahead == t )
     lookahead = System.in.read();
  else throw new Error("syntax error");
}
}
```

Funkcja Parser o takiej samej nazwie jak klasa, jest konstruktorem.
Jest wywoływana automatycznie, gdy obiekt klasy jest tworzony.
Konstruktor Parser inicjalizuje zmienną `lookahead` przez wczytanie tokena.

Klasa Parser w Javie

```
import java.io.*;

class Parser {
  static int lookahead;

  public Parser() throws IOException {
    lookahead = System.in.read();
  }

  void expr() {
    term();
    while ( true ) {
    if ( lookahead == '+' ) {
      match('+'); term();
      System.out.write('+');
      continue;
    }
    else if (lookahead == '-') {
       match('-'); term();
       System.out.write('-');
       continue;
    }
    else return;
  }
```

Funkcja expr realizuje nieterminale *expr* and *rest*

```
void term() throws IOException {
  if (Character.isDigit((char)lookahead){
     System.out.write((char)lookahead);
     match(lookahead);
  }
  else throw new Error("syntax error");
}

void match(int t) throws IOException {
  if ( lookahead == t )
     lookahead = System.in.read();
  else throw new Error("syntax error");
}
}
```

Klasa Parser w Javie

```
import java.io.*;

class Parser {
  static int lookahead;

  public Parser() throws IOException {
    lookahead = System.in.read();
  }

  void expr() {
    term();
    while ( true ) {
    if ( lookahead == '+' ) {
      match('+'); term();
      System.out.write('+');
      continue;
    }
    else if (lookahead == '-') {
       match('-'); term();
       System.out.write('-');
       continue;
    }
    else return;
  }
```

expr wywołuje funkcje term, a następnie ma pętlę while, która sprawdza, czy Lookahead pasuje albo do "+" lub do "-".

```
void term() throws IOException {
  if (Character.isDigit((char)lookahead){
     System.out.write((char)lookahead);
     match(lookahead);
  }
  else throw new Error("syntax error");
}

void match(int t) throws IOException {
  if ( lookahead == t )
     lookahead = System.in.read();
  else throw new Error("syntax error");
}
}
```

Klasa Parser w Javie

```
import java.io.*;

class Parser {
  static int lookahead;

  public Parser() throws IOException {
    lookahead = System.in.read();
  }

  void expr() {
    term();
    while ( true ) {
    if ( lookahead == '+' ) {
      match('+'); term();
      System.out.write('+');
      continue;
    }
    else if (lookahead == '-') {
       match('-'); term();
       System.out.write('-');
       continue;
    }
    else return;
  }
```

Funkcja term korzysta z funkcji isDigit należącej do klasy Character języka Java aby sprawdzić, czy symbol `lookahead` jest cyfrą.

```
void term() throws IOException {
  if (Character.isDigit((char)lookahead)){
     System.out.write((char)lookahead);
     match(lookahead);
  }
  else throw new Error("syntax error");
}

void match(int t) throws IOException {
  if ( lookahead == t )
     lookahead = System.in.read();
  else throw new Error("syntax error");
}
}
```

Klasa Parser w Javie

```
import java.io.*;

class Parser {
  static int lookahead;

  public Parser() throws IOException {
    lookahead = System.in.read();
  }

  void expr() {
    term();
    while ( true ) {
    if ( lookahead == '+' ) {
      match('+'); term();
      System.out.write('+');
      continue;
    }
    else if (lookahead == '-') {
       match('-'); term();
       System.out.write('-');
       continue;
    }
    else return;
  }
```

Konstrukcja (char) lookahead konwertuje lookahead na znak ponieważ lookahead jest zdeklarowana jako zmienna całkowita

```
void term() throws IOException {
  if (Character.isDigit((char)lookahead)){
     System.out.write((char)lookahead);
     match(lookahead);
  }
  else throw new Error("syntax error");
}

void match(int t) throws IOException {
  if ( lookahead == t )
     lookahead = System.in.read();
  else throw new Error("syntax error");
}
}
```

Klasa Parser w Javie

```
import java.io.*;

class Parser {
  static int lookahead;

  public Parser() throws IOException {
    lookahead = System.in.read();
  }

  void expr() {
    term();
    while ( true ) {
    if ( lookahead == '+' ) {
      match('+'); term();
      System.out.write('+');
      continue;
    }
    else if (lookahead == '-') {
       match('-'); term();
       System.out.write('-');
       continue;
    }
    else return;
  }
```

Funkcja match sprawdza terminale; odczytuje następny terminal jeśli symbol lookahead jest dopasowany do t, inaczej sygnalizuje błąd, wykonując throw new Error(„syntax error") ;

```
void term() throws IOException {
  if (Character.isDigit((char)lookahead)){
     System.out.write((char)lookahead);
     match(lookahead);
  }
  else throw new Error("syntax error");
}

void match(int t) throws IOException {
  if ( lookahead == t )
     lookahead = System.in.read();
  else throw new Error("syntax error");
 }
}
```

Klasa Parser w Javie

```
import java.io.*;

class Parser {
  static int lookahead;

  public Parser() throws IOException {
    lookahead = System.in.read();
  }

  void expr() {
    term();
    while ( true ) {
    if ( lookahead == '+' ) {
      match('+'); term();
      System.out.write('+');
      continue;
    }
    else if (lookahead == '-') {
       match('-'); term();
       System.out.write('-');
       continue;
    }
    else return;
  }
```

```
void term() throws IOException {
  if (Character.isDigit((char)lookahead){
     System.out.write((char)lookahead);
     match(lookahead);
  }
  else throw new Error("syntax error");
}

void match(int t) throws IOException {
  if ( lookahead == t )
     lookahead = System.in.read();
  else throw new Error("syntax error");
 }
}
```

Ten kod tworzy nowy wyjątek klasy Error

Program w języku C

Program w C

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

int lookahead;

void error() {
  printf("syntax error\n");
  exit(EXIT_FAILURE);
}

void match(int t) {
  if (lookahead == t)
    lookahead = getchar();
  else
    error();
}
```

Kolejność wołania funkcji określa lewa strona produkcji:

expr → *term* *rest*

rest → + *term* { print('+') }
rest
| – *term* { print('-') }
rest
| ε

term → **0** { print('0') }
term → **1** { print('1') }
…
term → **9** { print('9') }

```
void term () {
  if (isdigit(lookahead)) {
    putchar(lookahead);
    match(lookahead);
  }
  else
    error();
}

void rest() {
  if (lookahead == '+') {
    match('+'); term(); putchar('+'); rest();
  }
  else if (lookahead == '-') {
    match('-'); term(); putchar('-'); rest();
  }
  else
    ;
}
```

Program w C

```
void expr() {
  term(); rest();
}

int main(void) {
  lookahead = getchar();
  expr();
  putchar('\n');
  return EXIT_SUCCESS;
}
```

Kolejność wołania funkcji określa lewa strona produkcji:

$expr \rightarrow term \ \ rest$

$rest \rightarrow + \ term$ { print('+') } $rest$

$\quad | \ - \ term$ { print('-') } $rest$

$\quad | \ \varepsilon$

$term \rightarrow \mathbf{0}$ { print('0') }

$term \rightarrow \mathbf{1}$ { print('1') }

…

$term \rightarrow \mathbf{9}$ { print('9') }

Program w C

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

int lookahead;

void error() {
  printf("syntax error\n");
  exit(EXIT_FAILURE);
}

void match(int t) {
  if (lookahead == t)
    lookahead = getchar();
  else
    error();
}
```

Kolejność wołania funkcji określa lewa strona produkcji:

expr → *term* *rest*

rest → + *term* { print(‘+’) } *rest*

| – *term* { print(‘-’) } *rest*

| ε

term → **0** { print(‘0’) }

term → **1** { print(‘1’) }

…

term → **9** { print(‘9’) }

```
void term () {
  if (isdigit(lookahead)) {
    putchar(lookahead);
    match(lookahead);
  }
  else
    error();
}

void rest() {
  if (lookahead == '+') {
    match('+'); term(); putchar('+'); rest();
  }
  else if (lookahead == '-') {
    match('-'); term(); putchar('-'); rest();
  }
  else
    ;
}
```

zwraca wartość różną od zera gdy argument, który został przekazany do funkcji jest cyfrą.

Program w C, cd

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

int lookahead;

void error() {
  printf("syntax error\n");
  exit(EXIT_FAILURE);
}

void match(int t) {
  if (lookahead == t)
    lookahead = getchar();
  else
    error();
}
```

Kolejność wołania funkcji określa lewa strona produkcji:

expr → *term* *rest*

rest → + *term* { print(‘+’) } *rest*
| – *term* { print(‘-’) } *rest*
| ε

term → **0** { print(‘0’) }
term → **1** { print(‘1’) }
…
term → **9** { print(‘9’) }

```
void term () {
  if (isdigit(lookahead)) {
    putchar(lookahead);
    match(lookahead);
  }
  else
    error();
}

void rest() {
  if (lookahead == '+') {
    match('+'); term(); putchar('+'); rest();
  }
  else if (lookahead == '-') {
    match('-'); term(); putchar('-'); rest();
  }
  else
    ;
}
```

wysyła znak na standardowe wyjście, implementuje akcje { print(‘+’) }

Zastosowanie analizy wstępującej w oparciu o Lex i Yacc

Zastosowanie Lex'a i Yacc'a

```
Lex file:
%{
 #include <stdlib.h>
 #include "y.tab.h"
  void yyerror(char *); %}
 VAR [a-z]
%%
 {VAR} {yylval.value[0]=*yytext;
  yylval.value[1]='\0';
   return VAR;}
  [-+*/"^"\n] return *yytext;
  [ \t] ;
  yyerror("Invalid Char");
  %%
  int yywrap(void)
  { return 1; }
```

[a-z] = a | b | c | … | z,
Definiuje terminal VAR

Określa wartości atrybutów tokenów za pomocą odpowiedniego pola unii, zdeklarowanej na następnym slajdzie

Zastosowanie Lex'a i Yacc'a

- Yacc file:
- %{
- void yyerror(char *);
- int yylex(void);
- #include<stdio.h>
- char prefix[300];
- char postfix[300];
- char temp[52];
- %}
- %union { char value[300]; }
- %token <value> VAR
- %type <value> exp term fact

• w celu zadeklarowania typów symboli należy najpierw zdefiniować typ atrybutów za pomocą deklaracji %union

Definiuje symbole terminalne

Definiuje symbole nieterminalne

Zastosowanie Lex’a i Yacc’a

```
%%
program: exp '\n' { printf("%s\n",$1); }
               ;
exp: term { strcpy($$,$1); }
       | exp '+' term { /*strcat ($$,$1);*/
                        strcat ($$,$3);
                        strcat ($$,"+"); }
       | exp '-' term { /*strcat ($$,$1)*/;
                         strcat ($$,$3);
                         strcat ($$,"-"); }
       ;
```

kopiuje tekst z $1 do $$.

dopisuje tekst z $$ na koniec tekstu w $3.

Zastosowanie Lex’a i Yacc’a

```
term: fact { strcpy($$,$1); }
        | term '*' fact { strcat ($$,$3);
                          strcat ($$,"*"); }
        | term '/' fact { strcat ($$,$3);
                          strcat ($$,"/"); }
fact: VAR { strcpy($$,$1); }
      |  fact '^' VAR { strcat ($$,$3);
                        strcat($$,"^"); }
%%
```

Zastosowanie Lex’a i Yacc’a

```
void yyerror(char *s) {
        fprintf(stderr, "%s\n", s);
 }

int main(void) {
   yyparse();
   return 0;
}
```

Dziękuję za uwagę