Języki i Paradygmaty Programowania

Składnia

Marcin Benke

MIM UW

4 kwietnia 2016

Syntax is user interface

— Brendan Eich

Składnia i semantyka

- Składnia
  - Jakie są elementy języka
  - Jak można je komponować (by tworzyć programy lub zdania)
- Semantyka
  - Jakie jest *znaczenie* (poprawnego składniowo) programu/tekstu.
- Przykłady
  - Składnia: *Jakie słowa kluczowe są w języku?*
  - Składnia: *Co może wystąpić po lewej stronie przypisania*
  - Semantyka (statyczna): Czy można użyć liczby jako warunku **if**?
  - Semantyka: W jakiej kolejności $(+)$ oblicza swoje argumenty?

Analiza składniowa

Celem analizy składniowej jest budowa drzewa struktury programu.

Dzisiaj:

- Opis składni: gramatyki, BNF, EBNF
- Wywody, drzewa wywodu
- Wieloznaczność
- Analiza top-down i bottom-up
- Analiza metodą LL(1)

Gramatyki

Formalnie, a gramatyka bezkontekstowa jest czwórką:

$$G = \langle T, N, S, P \rangle$$

gdzie

- T jest zbiorem symboli terminalnych (leksemów)
- N jest zbiorem symboli nieterminalnych (rozłącznym z T)
- $S \in N$ jest symbolem początkowym
- P jest zbiorem produkcji postaci $\alpha \to \beta$ gdzie $\alpha \in N, \ \beta \in (T \cup N)^*$

Konwencja: symbole nieterminalne oznaczamy zwykle wielkimi literami.

Gramatyka indukuje naturalną relację przepisywania $\to$ na zbiorze $(T \cup N)^*$: napisy mogą być przepisywane zgodnie z produkcjami.

BNF

Backus-Naur Form — notacja dla gramatyk bezkontekstowych

- Symbole terminalne są wyróżniane przez użycie cudzysłowów, lub innej czcionki.
- W produkcjach ::= zastępuje $\to$
- Można skrótowo zapisywać zbiory produkcji:

$$E ::= E + T \mid T$$

  zamiast

$$E ::= E + T$$
$$E ::= T$$

- Pierwsza produkcja wyznacza symbol startowy.

EBNF

- Extended BNF.
- Nawiasy klamrowe oznaczają dowolną ilość powtórzeń:
$$A ::= \{\alpha\}$$
odpowiada
$$A ::= \epsilon \mid \alpha A$$
- Nawiasy kwadratowe oznaczają elementy opcjonalne:
$$A ::= [\alpha]$$
odpowiada
$$A ::= \epsilon \mid \alpha$$
- Jest kilka wariantów EBNF...

EBNF — przykład

```
stmts ::= stmt { ';' stmt }
if-stmt ::= 'if' boolexpr
           'then' stmts
          [ 'else' stmts ]
          'endif'
```

Wywody

Rozważmy gramatykę

$$
\begin{aligned}
E &\to E + E \mid E - E \mid T \\
T &\to 0 \mid 1 \mid \ldots \mid 9
\end{aligned}
$$

Wyrażenie $1 - 2 + 3$ może być wywiedzione:

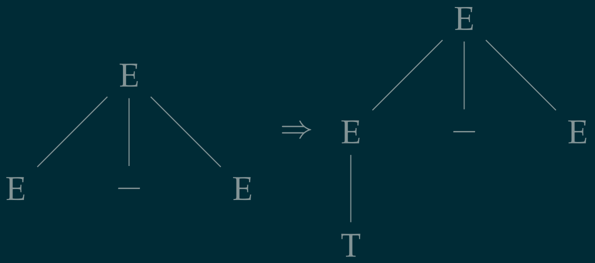
$$
\begin{aligned}
E &\Rightarrow E - E \\
&\Rightarrow T - E \\
&\Rightarrow 1 - E \\
&\Rightarrow 1 - E + E \\
&\Rightarrow 1 - T + E \\
&\Rightarrow 1 - 2 + E \\
&\Rightarrow 1 - 2 + T \\
&\Rightarrow 1 - 2 + 3
\end{aligned}
$$

Jest to wywód lewostronny — zawsze przepisujemy lewy nieterminal.

Drzewa wywodu

Wywód możemy przedstawić jako drzewo:

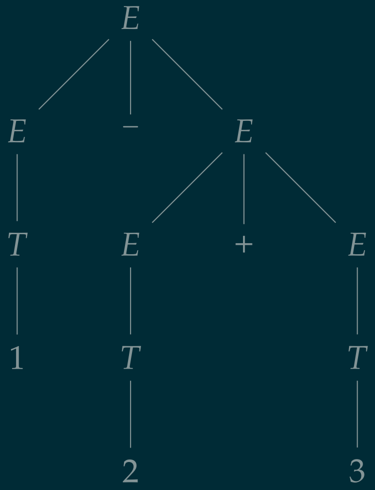
- Zaczynamy od drzewa złożonego z symbolu startowego;
- dla produkcji $A \to X_1 \ldots X_n$ zastępujemy A przez drzewo $A(X_1, \ldots, X_n)$



- wywiedzione słowo w liściach, od lewej do prawej;
- abstrahujemy od kolejnosci przepisywania nieterminali;
- jeśli dla danego słowa istnieje tylko jedno drzewo wywodu, to wywód tego słowa jest jednoznaczny.

Drzewa wywodu

Przykład: drzewo dla `1-2+3`:



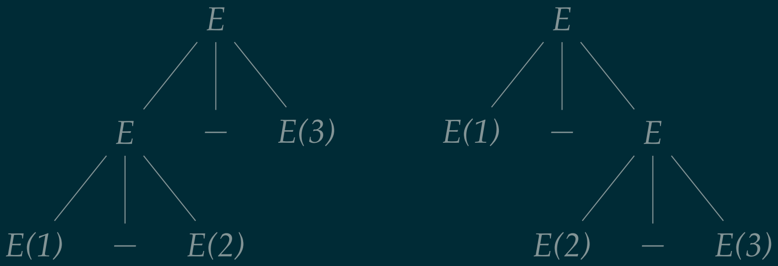
Drzewo wywodu reprezentuje *składnię konkretną*.

Wieloznaczność

Rozważmy gramatykę:

$$E ::= E - E \mid n$$

Istnieją dwa możliwe wywody `1-2-3`:



- Który wywód jest "poprawny"?
- '–' zwykle łączy w lewo, więc poprawny jest pierwszy wywód.
- Jak możemy uwzględnić ten fakt? Może trzeba nieco zmienić gramatykę?

Łączność

Możemy "usztywnić" gramatykę, tak aby istniał tylko jeden wywód; w przypadku

$$E ::= E - E \mid n$$

możemy wybrać albo prawe albo lewe E dla drugiego — w naszym przykładzie.

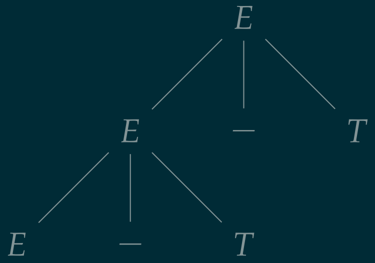
Chcemy wymusić aby wywód "wyliczył" wszystkie minusy w kolejności występowania

$$E ::= E - n \mid n$$

Łączność

Otrzymujemy następujące wyprowadzenie:

$$E \Rightarrow E - T \Rightarrow E - T - T \Rightarrow T - T - T \Rightarrow \dots$$



Dla operatora wiążącego w prawo, np potęgowania

$$E ::= T \uparrow E \mid T$$

Priorytety operatorów

Rozważmy:

$$
\begin{aligned}
E ::= {} & E + T \\
& \mid E * T \\
& \mid T \\
& \ldots
\end{aligned}
$$

Jak możemy opisać (w gramatyce), że '*' wiąże silniej niż '+'?

Symbol o niższym priorytecie musi być "bliżej" symbolu startowego:

$$
\begin{aligned}
E & ::= E + T \mid T \\
T & ::= T * F \mid F \\
F & ::= \ldots
\end{aligned}
$$

Wieloznaczność if-then-else

Rozważmy gramatykę:

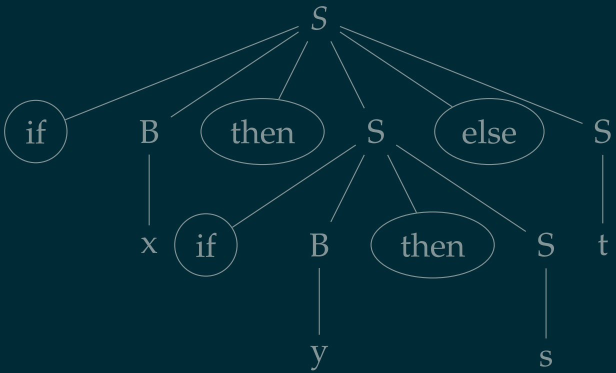
```
S ::= if B then S else S
    | if B then S
    | ...
```

wtedy konstrukcja

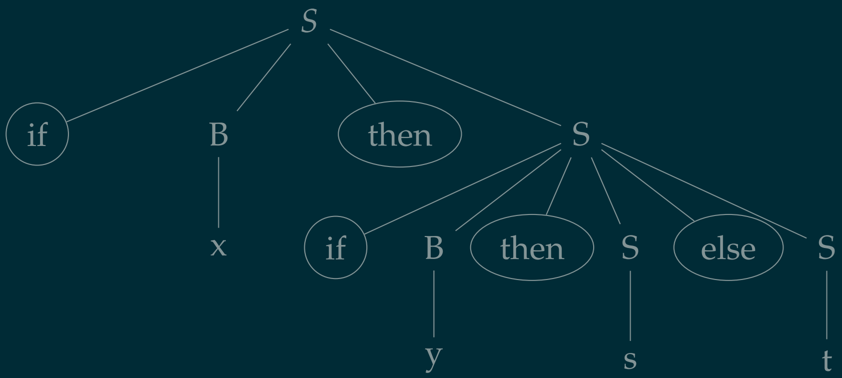
```
if x then if y then s else t
```

Ma dwa możliwe wywody:

Wieloznaczność if-then-else



Wieloznaczność if-then-else



Wieloznaczność if-then-else

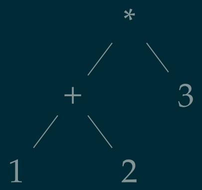
- Możemy rozwiązać problem zmieniając składnię języka:

```
S ::=
    if B then S else S fi
  | if B then S fi
  | ...
```

- Alternatywnie, możemy rozwiązać problem ad hoc mówiąc, że `else S` zawsze łączy się z najbliższym `if`.
- Można to wyrazić w gramatyce, ale jest to nieco kłopotliwe.

Składnia abstrakcyjna

- Drzewa wywodu są czasem nazywane *drzewami składni konkretnej*
- Zawierają informacje zbędne na dalszych etapach kompilacji
- Zamiast tego potrzebujemy drzew struktury zawierających tylko *informacje semantyczne* (opisujące znaczenie programu).
- Drzewo struktury dla `(1 + 2) * 3`:



Nawiasy należą do składni konkretnej, ale nie do abstrakcyjnej.

Składnia abstrakcyjna

Składnia abstrakcyjna to zbiór typów pozwalających reprezentować strukturę programu.

Abstrahujemy od elementów takich jak nawiasy, znaki przestankowe, ...

Na przykład:

```
Stmt  = SWhile Exp Stmt
      | SAssign Var Exp
      | SReturn Exp
      | SBlock [Stmt]
```

Analiza syntaktyczna

Analizator syntaktyczny (*parser*, od łacińskiego *pars orationis*), jest funkcją dla danego słow dającą drzewo struktury lub komunikat o błędzie.

- Algorytm Youngera (CYK): $\mathcal{O}(n^3)$
- Istnieją efektywne algorytmy dla pewnych klas gramatyk.

Dwa zasadnicze podejścia:

- Top-down: próbujemy sparsować określoną konstrukcję (nieterminal); drzewo struktury budowane od korzenia do liści.
- Bottom-up: w danym napisie znajdujemy możliwe konstrukcje; drzewo budowane od liści do korzenia ze znalezionych kawałków.

Analiza syntaktyczna top-down

Schemat analizy top-down możem zapisać jako automat:

- Jeden stan, alfabet stosowy $\Gamma = N \cup T$, akceptacja pustym stosem.
- Na szczycie stosu $a \in T$:
  - jeśli na wejściu a — zdejmij ze stosu, wczytaj następny symbol.
  - wpp — błąd: oczekiwano a.
- Na szczycie stosu $A \in N$, na wejściu a:
  - wybieramy produkcję $A \to \alpha$
  - na stosie zastępujemy A przez α

Dla automatu deterministycznego, wybór produkcji jest ważny; zbiór symboli dla których wybieramy produkcję $A \to \alpha$ nazywamy $SELECT(A \to \alpha)$.

Produkcje postaci $A \to A\beta$ prowadzą do zapętlenia bez postepu na wejściu.

Kombinatory parsujące

Niedeterminizm można reprezentować przez monadę list, np.

```
newtype Parser a = Parser (String -> [(a,String)])
```

albo, używając transformatorów monad

```
type Parser a = StateT String (ErrorT String []) a
```

oraz kombinatory (funkcje) reprezentujące elementarne parsery
i sposoby łączenia parserów:

```
item :: Parser Char
(<|>) :: Parser a -> Parser a -> Parser a
sat :: (Char->Bool) -> Parser Char
char :: Char -> Parser Char
char x = sat (==x)
many, many1 :: Parser a -> Parser [a]
```

Kombinatory parsujące

Przy użyciu tych kombinatorów mozemy np. dla gramatyki

```
Int :: Nat | '-' Nat
Nat ::= {digit}
```

napisać parser(y):

```
pInt, pNat :: Parser Integer
pInt =  pNat <|> negative pNat  where
  negative :: (Num a) => Parser a -> Parser a
  negative p = fmap negate (char '-' >> p)

pNat = fmap (foldl (\x y -> 10*x+toInteger y) 0)
            pDigits
```

Ograniczanie niedeterminizmu

Ponieważ pełny niedeterminizm może prowadzić do eksplozji złożoności, operator wyboru `<|>` można uczynić deterministycznym.

Wybieramy alternatywę podglądając tylko pierwszy symbol wejścia.

```
many, many1 :: Parser a -> Parser [a]
many p = many1 p <|> return []

many1 p = do { a <- p; as <- many p; return (a:as)}
```

Zauważmy, że kolejność argumentów operatora wyboru ma teraz znaczenie; **many** tak jak jest zdefiniowane da nam *najdłuższe* możliwe dopasowanie (przeważnie tego właśnie chcemy).

Niedeterministyczny wybór mozna odzyskać przy pomocy **try**

Parsec

Pakiet **parsec**: biblioteka kombinatorów parsujących

```
import Text.ParserCombinators.Parsec

-- type Parser a = GenParser Char () a
-- parse :: GenParser tok () a -> SourceName -> [tok]
--          -> Either ParseError a

run :: Parser a -> [Char] -> Either ParseError a
run p s = parse p "(interactive)" s

pInt :: Parser Integer
pInt = negative pNat <|> pNat where
  negative :: (Num a) => Parser a -> Parser a
  negative p = fmap negate (char '-' >> p)
...
```

Parsec — wyrażenia arytmetyczne

```haskell
pExp, pT, pF :: Parser Exp
pExp = pT `chainl1` addop
pT = pF `chainl1` mulop
pF = ENum <$> integer <|> parens pExp

addop =  do{ symbol "+"; return EAdd }
     <|> do{ symbol "-"; return ESub }

data Exp = ENum Integer
         | EAdd Exp Exp | ESub Exp Exp
         | EMul Exp Exp | EDiv Exp Exp
```

Parsec — inny przykład

```
pFunc = do
  reserved "func"
  params <- parens pParams
  body <- pExp
  return (EFunDecl params body))

pParams = pParam `sepBy` (symbol ",")
pParam = identifier

pCall f = do
  symbol "("
  es <- pActParams
  symbol ")"
  return $ EFunCall f es
pActParams = pExp `sepBy` (symbol ",")
```

Zbiory *FIRST*

Notacja

Niech $w \in T^*$

$$k : w = \begin{cases} a_1 a_2 \ldots a_k, & \text{jeśli } w = a_1 a_2 \ldots a_k v \\ w\#, & \text{jeśli } |w| < k. \end{cases}$$

(pierwszych k znaków słowa w)

Definicja (*FIRST*)

Niech $w \in (T \cup N)^*$.

$$FIRST(w) = \{\alpha : \exists \beta \in T^*,\ w \rightarrow^* \beta,\ \alpha = 1 : \beta\}$$

(pierwsze znaki słów wyprowadzalnych z w).

Zbiory *FOLLOW*

Definicja (*FOLLOW*)

Niech $A \in N$

$$FOLLOW(A) = \{\alpha : \exists \beta \in T^*,\ S \to^* \mu A \beta,\ \alpha = 1 : \beta\}$$

(znaki mogące wystąpić za A).

Zbiory SELECT

$$SELECT(A \to \alpha) = FIRST(\alpha \cdot FOLLOW_k(A))$$

Inaczej mówiąc:

- jeśli $\alpha \to^* \epsilon$, to

$$SELECT(A \to \alpha) = FIRST'(\alpha) \cup FOLLOW(A)$$

- jeśli $\alpha \not\to^* \epsilon$

$$SELECT(A \to \alpha) = FIRST(\alpha) = FIRST'(\alpha)$$

gdzie $FIRST'(\alpha) = FIRST(\alpha) \setminus \{\#\}$

Gramatyka jest LL(1), jeśli dla każdej pary (różnych) produkcji $A \to \alpha$, $A \to \beta$ ich zbiory $SELECT$ są rozłączne.

Dla gramatyki LL(1) możemy łatwo skonstruować deterministyczny parser.

Problemy

Gramatyka nie jest LL(1) jeśli zawiera zbiory produkcji postaci

$$A \to \alpha\beta \mid \alpha\gamma$$

lub produkcje postaci

$$A \to A\beta$$

W drugim przypadku parser się nie zatrzyma!

Gramatykę, w której występują te problemy możemy często przekształcić
do równoważnej gramatyki LL(1).

Lewostronna faktoryzacja

Problem pierwszego rodzaju możemy rozwiązać "wyłączając przed nawias" wspólne początki produkcji:

$$A \to \alpha\beta \mid \alpha\gamma$$

zastępujemy przez

$$\begin{aligned} A &\to \alpha Z \\ Z &\to \beta \mid \gamma \end{aligned}$$

gdzie Z jest świeżym nieterminalem.

Eliminacja lewostronnej rekursji

Zbiór produkcji

$$A \to A\alpha \mid \beta$$

zastępujemy

$$\begin{aligned} A &\to \beta R \\ R &\to \alpha R \mid \varepsilon \end{aligned}$$

Na przykład, dla gramatyki

$$E \to E + T \mid T$$

otrzymujemy gramatykę

$$\begin{aligned} E &\to TR \\ R &\to +TR \mid \varepsilon \end{aligned}$$

Wyliczanie *FIRST*

Notacja: $FIRST'(w) = FIRST(w) \setminus \{\#\}$

Dla $t \in T$ mamy $FIRST(t) = \{t\}$.
Dla $A \in N$ mamy:
$A \to \alpha_1 \mid \ldots \mid \alpha_n \Rightarrow FIRST(A) \supseteq FIRST(\alpha_1) \cup \ldots \cup FIRST(\alpha_n)$
Dla $A \to X_1 \ldots X_n$

- $FIRST(A) \supseteq FIRST'(X_1)$
- $X_1 \to^* \varepsilon \Rightarrow FIRST(A) \supseteq FIRST'(X_2)$
- $X_1 X_2 \to^* \varepsilon \Rightarrow FIRST(A) \supseteq FIRST'(X_3)$
- $\ldots$
- $X_1 X_2 \ldots X_n \to^* \varepsilon \Rightarrow \# \in FIRST(A)$

Prosty algorytm: zgodnie z powyższymi regułami powiększamy zbiory FIRST tak długo, jak któryś ze zbiorów się powiększa (obliczamy najmniejszy punkt stały).

Proste wyliczanie zbiorów FIRST

```
first :: Grammar -> Symbols -> Symbols
first g sy = maybeAddEot $ first' g sy where
  maybeAddEot ss | nullable g sy = EOT:ss
                 | otherwise = ss
first' :: Grammar -> Symbols -> Symbols
first' g sy = go [] sy where
  go _ []                             = []
  go v (h:t) | h `is_terminal` g = [h]
     | nullable_nt g h    = go (h:v) t ++ goNT v h
     | otherwise          = goNT v h
  goNT v h
     | h `elem` v = [] -- go v t
     | otherwise = go (h:v) =<< rhs_nt g h
rhs_nt :: Grammar -> Symbol -> [Symbols]
-- lista prawych stron produkcji dla danego symbolu
```

Wyliczanie *FOLLOW*

Dla każdych $A, X \in N$, $a \in T$, $\alpha, \beta \in (N \cup T)^*$ mamy:

- $A \to \alpha X a \beta \in P$, to $a \in FOLLOW(X)$.
- $A \to \alpha X \in P$, to $FOLLOW(A) \subseteq FOLLOW(X)$
- $A \to \alpha X \beta \in P$, to $FIRST'(\beta) \subseteq FOLLOW(X)$
- $A \to \alpha X \beta \in P$, $\beta \to^* \varepsilon$ to $FOLLOW(A) \subseteq FOLLOW(X)$
- $\# \in FOLLOW(S)$ dla symbolu startowego S.

Prosty algorytm: zgodnie z powyższymi regułami powiększamy zbiory FOLLOW tak długo, jak któryś ze zbiorów się powiększa (obliczamy najmniejszy punkt stały).

Przykład

$$
\begin{array}{ll}
E \to E + T & \textsc{Select}(E \to E + T) = \textsc{First}(E) = \textsc{First}(T) \\
E \to T & \textsc{Select}(E \to T) = \textsc{First}(T) = \textsc{First}(F) \\
\hline
T \to T * F & \textsc{Select}(T \to T * F) = \textsc{First}(T) = \textsc{First}(F) \\
T \to F & \textsc{Select}(T \to F) = \textsc{First}(F) = \{(, \mathbf{a}\} \\
\hline
F \to (E) & \\
F \to \mathbf{a} &
\end{array}
$$

Gramatyka nie jest LL(1).

Transformacja do postaci LL(1)

Usuwamy lewostronną rekursję:

$$E \to TE' \qquad E' \to +TE' \mid \varepsilon$$

$E \to TE'$	niepotrzebne Select
$E' \to +TE'$	$\textsc{Select}(E' \to +TE') = \{+\}$
$E' \to \varepsilon$	$\textsc{Select}(E' \to \varepsilon) = \textsc{Follow}(E') =$ $= \textsc{Follow}(E) = \{), \#\}$
$T \to FT'$	
$T' \to *FT'$	$\textsc{Select}(T' \to *FT') = \{*\}$
$T' \to \varepsilon$	$\textsc{Select}(T' \to \varepsilon) = \textsc{Follow}(T') = \mathit{Follow}(T)$ $= \textsc{First}(E' \cdot \mathit{Follow}(E')) = \{+,), \#\}$
$F \to (E)$	$\textsc{Select}(F \to (E)) = \{(\}$
$F \to \mathbf{a}$	$\textsc{Select}(F \to \mathbf{a}) = \{\mathbf{a}\}$