Abstract Syntax Tree Generation

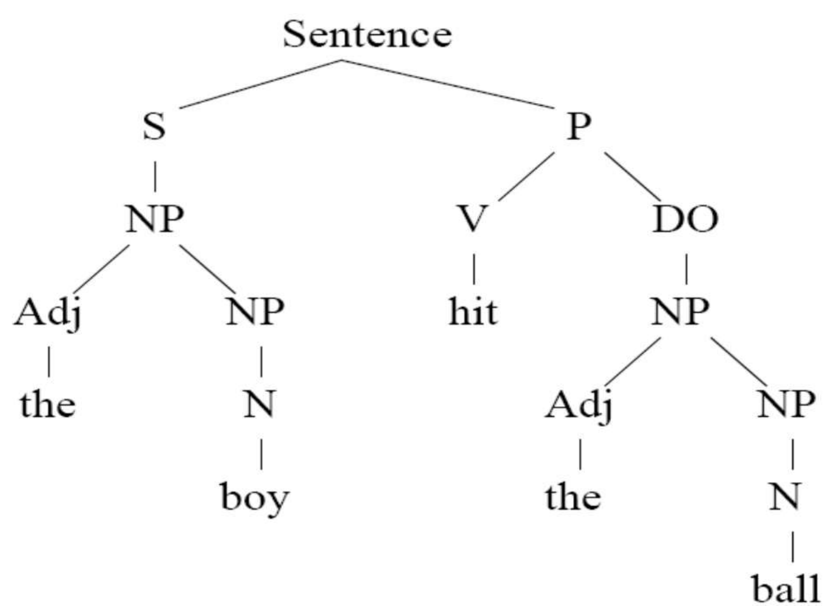
Programming Languages
Lecture 5

Adeesha Wijayasiri

Grammar:

$$
\begin{array}{ll}
\text{Sentence} & \rightarrow \text{S P} \\
\text{S} & \rightarrow \text{NP} \\
\text{P} & \rightarrow \text{V DO} \\
\text{DO} & \rightarrow \text{NP} \\
\text{NP} & \rightarrow \text{Adj NP} \\
 & \rightarrow \text{N} \\
\text{N} & \rightarrow \text{boy} \\
 & \rightarrow \text{ball} \\
\text{Adj} & \rightarrow \text{the} \\
 & \rightarrow \text{green} \\
\text{V} & \rightarrow \text{hit}
\end{array}
$$

DERIVATION TREE:



Building Derivation Trees

Sample Input :

```
- + i - i * ( i + i ) / i + i
```

derivation tree construction:

- Bottom-up.
- On each pass, scan entire expression, process operators with highest precedence (parentheses are highest).
- Lowest precedence operators are last, at the top of tree.

Grammar:

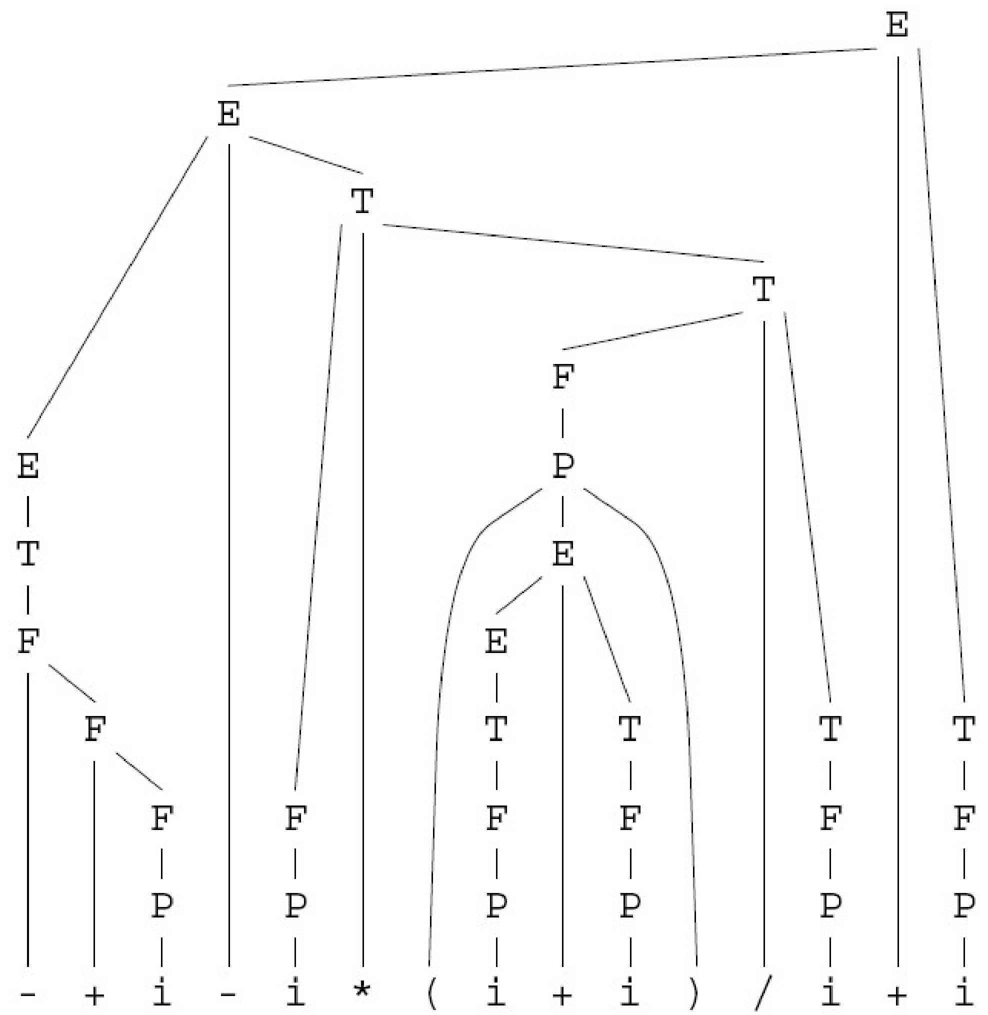
$$
\begin{array}{llll}
E \rightarrow E+T & T \rightarrow F*T & F \rightarrow -F & P \rightarrow (E) \\
\quad \rightarrow E-T & \quad \rightarrow F/T & \quad \rightarrow +F & \quad \rightarrow i \\
\quad \rightarrow T & \quad \rightarrow F & \quad \rightarrow P &
\end{array}
$$

- + i - i * (i + i) / i + i

Grammar:

$$
\begin{array}{llll}
E \rightarrow E+T & T \rightarrow F*T & F \rightarrow -F & P \rightarrow (E) \\
\quad \rightarrow E-T & \quad \rightarrow F/T & \quad \rightarrow +F & \quad \rightarrow i \\
\quad \rightarrow T & \quad \rightarrow F & \quad \rightarrow P &
\end{array}
$$

DERIVATION TREE:



Abstract Syntax Trees

- AST is a condensed version of the derivation tree.
- No noise (intermediate nodes).
- String-to-tree transduction grammar:
  - rules of the form $A \rightarrow \omega$ => 's'.

Example

E → E **+** T => **+**

→ E **-** T => **-**

→ T

T → F ***** T => *****

→ F **/** T => **/**

→ F

F → **-** F => `neg`

→ **+** F => **+**

→ P

P → '(' E ')'

→ `i` => `i`

Grammar:

$E \rightarrow E+T$ => +

$\rightarrow E-T$ => -

$\rightarrow T$

$T \rightarrow F*T$ => *

$\rightarrow F/T$ => /

$\rightarrow F$

$F \rightarrow -F$ => neg

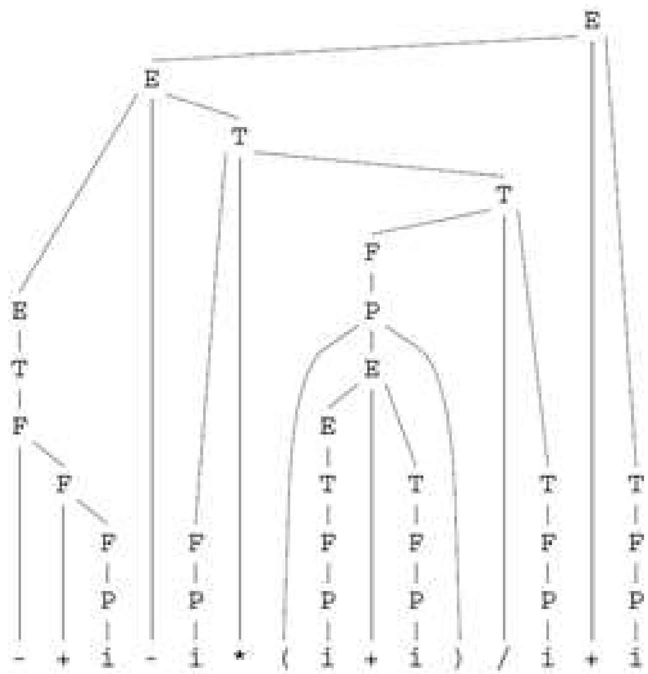
$\rightarrow +F$ => +

$\rightarrow P$

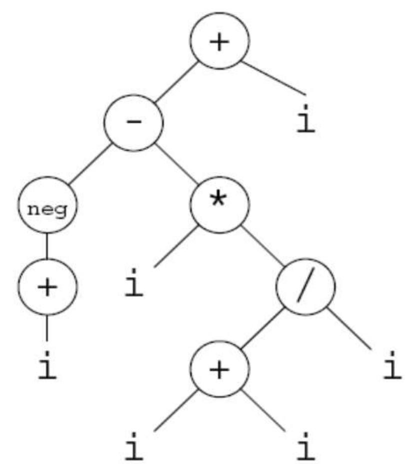
$P \rightarrow (E)$

$\rightarrow i$ => i

DERIVATION TREE:



ABSTRACT SYNTAX TREE:



Let's Build a Few Abstract Syntax Trees

- Example 1: Factorial.
- Example 2: Palindrome.
- Example 3: Add numbers from list.
- Example 4: Build tuple of pairs of characters.

Building AST's:

Example 1 : Factorial, Top-down, Counting from 1 to n.

```
let f n = rf n 1 1 where
   rec rf n c r = c eq n+1 -> r  | rf n (c+1) (c*r)
in Print (f 3, f 5, f 7)
```

Example 2: Palindrome.

```
let rec r s = s eq '' -> ''
        | Conc (r (Stern s)) (Stem s)
within
  p s = not Isstring s -> 'error'
        | s eq r s
in Print (p '1234', p 'abcba')
```

Example 3: Add numbers from list of arguments.

```
let add n = radd 0 n
   where rec radd r n = n eq 0 -> r
      | radd (r+n)
in Print (add 2 3 4 0)
```

Example 4 Build a tuple of pairs of characters.

```
let rec Rev S =
     S eq '' -> ''
     | (Rev(Stern S)) @Conc (Stem S )
within
     Pairs (S1,S2) =
      not (Isstring S1 & Isstring S2)
        -> 'both args not strings'
        | P (Rev S1, Rev S2)
           where rec P (S1, S2) =
             S1 eq '' & S2 eq ''
              -> nil
              | (Stern S1 eq '' & Stern S2 ne '') or
                (Stern S1 ne '' & Stern S2 eq '')
                -> 'unequal length strings'
              | (P (Stern S1, Stern S2)
                 aug ((Stem S1) @Conc (Stem S2)))
 in Print ( Pairs ('abc','def'))
```

YE COMPLEAT RPAL SPECIFICATION

(or, the Itty Bitty Book of RPAL)

RPAL's LEXICON:

```
Identifier -> Letter (Letter | Digit | '_')*                    => '<IDENTIFIER>'

Integer    -> Digit+                                             => '<INTEGER>';

Operator   -> Operator_symbol+                                   => '<OPERATOR>';

String     -> ''''
              ( '\' 't' | '\' 'n' | '\' '\' | '\' ''''
              | '('     | ')'     | ';'     | ','
              | ' '
              | Letter | Digit | Operator_symbol
              )* ''''                                            => '<STRING>';

Spaces     -> ( ' ' | ht | Eol )+                                => '<DELETE>';

Comment    -> '//'
             ( '''' | '(' | ')' | ';' | ',' | '\' | ' '
              | ht | Letter | Digit | Operator_symbol
             )* Eol                                              => '<DELETE>';

Punction   -> '('                                                => '('
           -> ')'                                                => ')'
           -> ';'                                                => ';'
           -> ','                                                => ',';

Letter     -> 'A'..'Z' | 'a'..'z';

Digit      -> '0'..'9';

Operator_symbol
           -> '+' | '-' | '*' | '<' | '>' | '&' | '.'
            | '@' | '/' | ':' | '=' | '~' | '|' | '$'
            | '!' | '#' | '%' | '^' | '_' | '[' | ']'
            | '{' | '}' | '"' | '`' | '?';
```

RPAL's Phrase Structure Grammar:

```
# Expressions ############################################

E     -> 'let' D 'in' E                          => 'let'
      -> 'fn'  Vb+ '.' E                         => 'lambda'
      ->  Ew;
Ew    -> T  'where' Dr                           => 'where'
      -> T;

# Tuple Expressions ######################################

T     -> Ta ( ',' Ta )+                          => 'tau'
      -> Ta ;
Ta    -> Ta 'aug' Tc                             => 'aug'
      -> Tc ;
Tc    -> B '->' Tc '|' Tc                        => '->'
      -> B ;

# Boolean Expressions ####################################

B     -> B 'or' Bt                               => 'or'
      -> Bt ;
Bt    -> Bt '&' Bs                               => '&'
      -> Bs ;
Bs    -> 'not' Bp                                => 'not'
      -> Bp ;
Bp    -> A ('gr' | '>' ) A                       => 'gr'
      -> A ('ge' | '>=') A                       => 'ge'
      -> A ('ls' | '<' ) A                       => 'ls'
      -> A ('le' | '<=') A                       => 'le'
      -> A 'eq' A                                => 'eq'
      -> A 'ne' A                                => 'ne'
      -> A ;
```

```
# Arithmetic Expressions ##################################

A     -> A '+' At                              => '+'
      -> A '-' At                              => '-'
      ->   '+' At
      ->   '-' At                              => 'neg'
      -> At ;
At    -> At '*' Af                             => '*'
      -> At '/' Af                             => '/'
      -> Af ;
Af    -> Ap '**' Af                            => '**'
      -> Ap ;
Ap    -> Ap '@' '<IDENTIFIER>' R               => '@'
      -> R ;

# Rators And Rands ########################################

R     -> R Rn                                  => 'gamma'
      -> Rn ;
Rn    -> '<IDENTIFIER>'
      -> '<INTEGER>'
      -> '<STRING>'
      -> 'true'                                => 'true'
      -> 'false'                               => 'false'
      -> 'nil'                                 => 'nil'
      -> '(' E ')'
      -> 'dummy'                               => 'dummy'  ;
```

```
# Definitions ############################################

D     -> Da 'within' D                                    => 'within'
      -> Da ;
Da    -> Dr ( 'and' Dr )+                                 => 'and'
      -> Dr ;
Dr    -> 'rec' Db                                         => 'rec'
      -> Db ;
Db    -> Vl '=' E                                         => '='
      -> '<IDENTIFIER>' Vb+ '=' E                         => 'fcn_form'
      -> '(' D ')' ;

# Variables ##############################################

Vb    -> '<IDENTIFIER>'
      -> '(' Vl ')'
      -> '(' ')'                                          => '()';
Vl    -> '<IDENTIFIER>' list ','                          => ','?;
```

Thank You!

REFERENCES

- Programming Language Pragmatics by Michael L. Scott. 3rd edition. Morgan Kaufmann Publishers. (April 2009).
- Lecture Slides of Dr.Malaka Walpola and Dr.Bermudez