Replacing recursion with iteration

Adeesha Wijayasiri

Replacing recursion with iteration

- To make our grammar LL(1), we introduced nonterminals X,Y,Z. None are needed.

  SL isn’t needed, either.

Grammars (So FAR)

Our "model" PL grammar.

```
 S → begin SL end  {begin}
   → id := E;      {id}
SL → SL S          {begin,id}
   → S             {begin,id}
 E → E+T           {(,id}
   → T             {(,id}
 T → P*T           {(,id}
   → P             {(,id}
 P → (E)           {(}
   → id            {id}
```

Modified, LL(1)) grammar.

```
S    → begin SL end  {begin}
     → id := E ;     {id}
SL   → S Z           {begin,id}
Z    → S Z           {begin,id}
     →               {end}
E    → T Y           {(,id}
Y    → + T Y         {+}
     →               {;,)}
T    → P X           {(,id}
X    → * T           {*}
     →               {;,+,)}
P    → (E)           {(}
     → id            {id}
```

Procedures SL, X, Y and Z can all be eliminated.

Replacing recursion

```
proc S;
  case Next_Token of
       T_begin :      Read(T_begin);
                    ⎡ repeat
                    ⎢     S();
                    ⎣ until Next_Token ∉ {T_begin,T_id};
                      Read(T_end);
         T_id :       Read(T_id);
                      Read (T_:=);
                      E();
                      Read (T_;);
       otherwise      Error;
  end;
end;
```

Replaces call to SL, and recursion on Z.

```
S  → begin SL end
   → id := E ;
SL → S Z
 Z → S Z
   →
```

Regular Right-Part Grammar:

```
S → begin S+ end
  → id := E ;
```

Replacing recursion

E	→	**TY**
Y	→	**+TY**
	→	
T	→	**PX**
X	→	***T**
	→	

Replaces call to Y, and recursion on Y.

```
proc E;
       T();
       while Next_Token = T_+ do
         Read (T_+);
         T();
       od;
end;
```

Replaces call to X.

```
proc T;
       P();
       if Next_Token = T_*
         then  Read (T_*); T();
end;
```

Regular Right-Part Grammar:

E	→	**T(+T)***
T	→	**P(*T)?**

Replacing recursion

```
proc P;
  case  Next Token of
        T_(: Read(T_();
             E();
             Read(T_));
      T_id: Read(T_id);
       otherwise Error;
    end;
end;
```

No change!

Regular Right-Part Grammar:

```
S → begin S+ end
  → id := E ;
E → T(+T)*
T → P(*T)?
P → (E)
  → id
```

summary

- To make our grammar LL(1), we introduced nonterminals X,Y,Z.
- We just got rid of them.
- Got rid of SL, too.
- Resulting code is remarkably simple.

Bottom-up derivation tree, original grammar

Topics

Red: now

- Possibilities:
  - Derivation tree or Abstract Syntax Tree.
  - Top-down, or Bottom-up.
  - For original or modified grammar !
- Leading up to:

  AST, bottom-up, for the original grammar ("the one").

BU DT, original grammar

```
S  → begin SL end
   → id := E ;
SL → SL S
   → S
```

```
proc S;
  case Next_Token of
        T_begin :      Read(T_begin);
                       S();
                       Write (SL → S);
                       while Next_Token ∊ {T_begin,T_id} do
                            S();
                            Write (SL → SL S);
                       Read(T_end);
                       Write (S → begin SL end);
           T_id :      Read(T_id);
                       Read (T_:=);
                       E();
                       Read (T_;);
                       Write (S → id :=E ;);
        otherwise      Error;
  end;
end;
```

BU DT, original grammar

```
proc E;
        T(); Write (E → T);
        while Next_Token = T_+ do
          Read (T_+);
          T();
          Write (E → E+T);
        od;
end;

proc T;
        P();
        if Next_Token = T_*
                then    Read (T_*); T();
                        Write (T → P*T);
                else    Write (T → P);
end;
```

E	→	E+T
	→	T
T	→	P*T
	→	T

BU DT, original grammar

```
proc P;
  case  Next Token of
        T_(: Read(T_();
                E();
                Read(T_));
                Write (P → (E));
      T_id: Read(T_id);
                Write (P → id);
        otherwise Error;
    end;
end;
```

We combined:

- **Top-down parsing**
  - **LL(1) grammar**
  - **Pre-order process**
- **Bottom-up tree construction**
  - **Original grammar**
  - **Post-order process**

Parser output

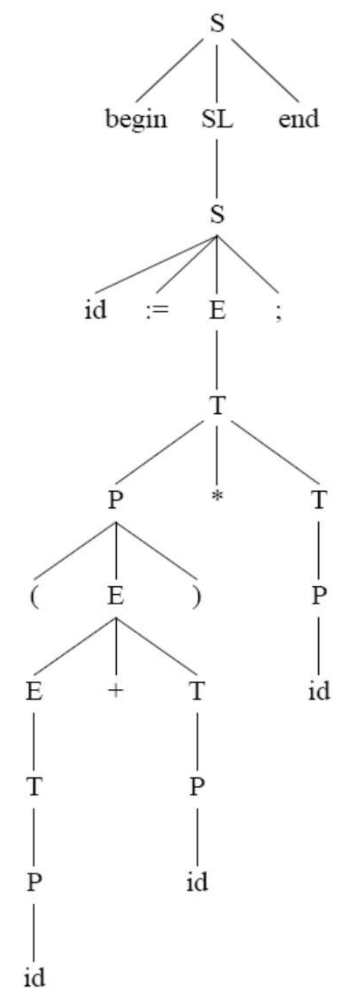
- Input String:

  begin id := (id + id) * id; end

- Output:

P → id
T → P
E → T
P → id
T → P
E → E+T
P → (E)
P → id
T → P

T → P*T
E → T
S → id:=E;
SL→ S
S → begin SL end



summary

Red: done

- Possibilities:
  - **Derivation tree** or Abstract Syntax Tree.
  - Top-down, or **Bottom-up**.
  - For **original** or modified grammar !
- Clean implementation, using stack of trees.
- ONE MORE THING TO DO: BUILD the AST ! ("the one")

Bottom-up AST, original grammar

Topics

Red: now

- Possibilities:
  - Derivation tree or **Abstract Syntax Tree**.
  - Top-down, or **Bottom-up**.
  - For **original** or modified grammar !
- OUR FINAL GOAL ! ("the one")
- Build the AST, for the original grammar, bottom-up.
- This is THE way to build a parser.

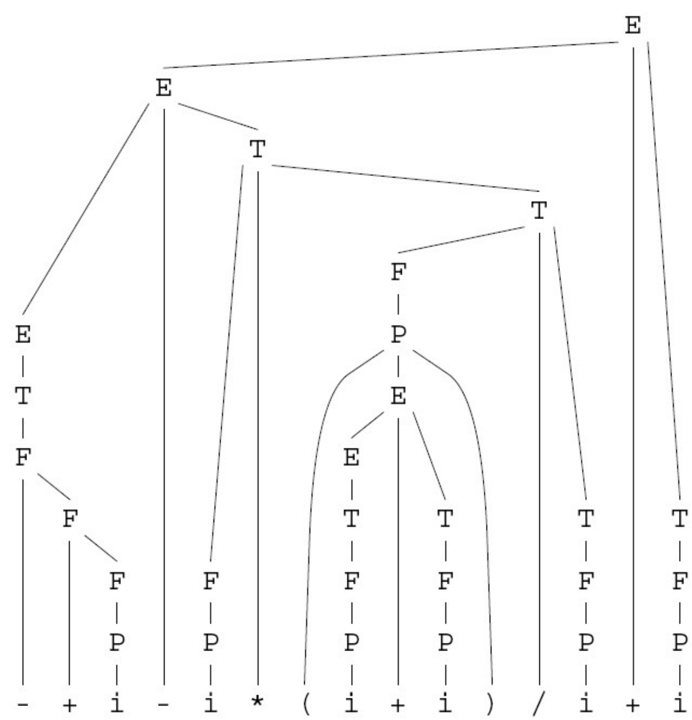
Building Derivation Trees

Sample Input : **- + 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

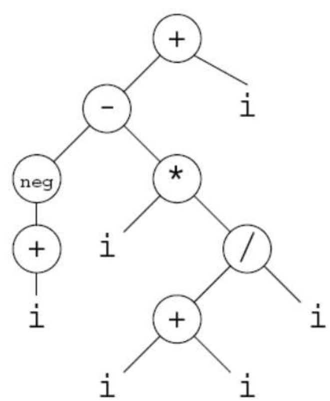
- AST is a condensed version of the derivation tree.
- No noise (intermediate nodes).
- String-to-tree transduction grammar:
  - rules of the form $A \rightarrow \omega$ => 's'.

Example

Grammar:

$$
\begin{aligned}
E &\rightarrow E+T && => + \\
  &\rightarrow E-T && => - \\
  &\rightarrow T && \\
T &\rightarrow F*T && => * \\
  &\rightarrow F/T && => / \\
  &\rightarrow F && \\
F &\rightarrow -F && => \text{neg} \\
  &\rightarrow +F && => + \\
  &\rightarrow P && \\
P &\rightarrow (E) && \\
  &\rightarrow i && => i
\end{aligned}
$$

ABSTRACT SYNTAX TREE:



AST, Bottom-UP, original grammar

```
S   → begin S+ end => 'block'
    → id := E ;    => 'assign'
```

```
proc S
   int N=1;
   case Next_Token of
         T_begin:Read(T_begin);
                 S();
                 while Next_Token ∊ {T_begin,T_id} do
                     S();
                     N++;
                 Read(T_end);
                 Build_tree('block', N);
            T_id:Read(T_id);
                 Read (T_:=);
                 E();
                 Read (T_;);
                 Build_tree('assign',2);
       otherwise Error;
   end;
end;
```

Build Tree ('x',n):

1. Pop n trees,
2. Build 'x' parent node,
3. Push new tree.

Read() no longer builds tree nodes, except for <id>, <int>, etc.

AST, Bottom-UP, original grammar

```
proc E;
        T();
        while Next_Token = T_+ do
          Read (T_+);
          T();
          Build_tree('+',2);
        od;
end;

proc T;
        P();
        if Next_Token = T_*
          then Read (T_*); T();
               Build_tree('*',,2);
end;
```

```
E → E+T => '+'
  → T
T → P*T => '*'
  → P
```

AST, Bottom-UP, original grammar

```
proc P;
  case  Next Token of
        T_(: Read(T_();
                 E();
                 Read(T_));
       T_id: Read(T_id);
       otherwise Error;
    end;
end;
```

```
P →(E)
  → id
```

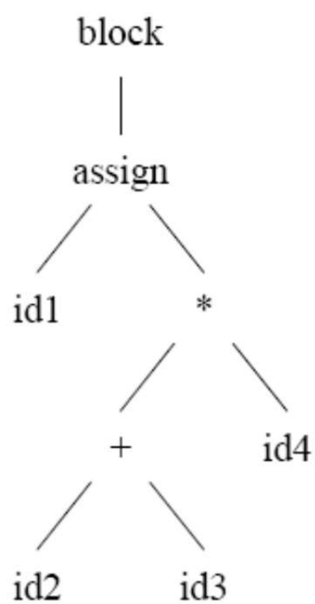
No Build_tree() necessary

Parser output

- Input String:

  begin $\mathbf{id_1} := (\mathbf{id_2} + \mathbf{id_3}) * \mathbf{id_4}$; **end**
- Output (Tree-building actions):

BT(id_1,0)
BT(id_2,0)
BT(id_3,0)
BT('+',2)
BT(id_4,0)
BT('*',2)
BT('assign',2)
BT('block,1)




```
S → begin S+ end  => 'block'
  → id := E ;     => 'assign'
E → E+T           => '+'
  → T
T → P*T           => '*'
  → P
P →(E)
  →id
```

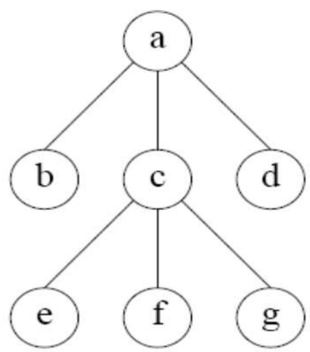
How to write a parser

- Starting point:
  - A Regular Right-Part, Syntax-Directed Translation Scheme
- Write parser directly from the grammar.
- There’s (likely) an LL(1) grammar lurking in there, but
  - Don’t need to write it explicitly.
- Calculate Select sets, er, well … selectively.
- Don’t need Derivation Tree.
- Recognize patterns, build code.
- This is THE way to build a parser.

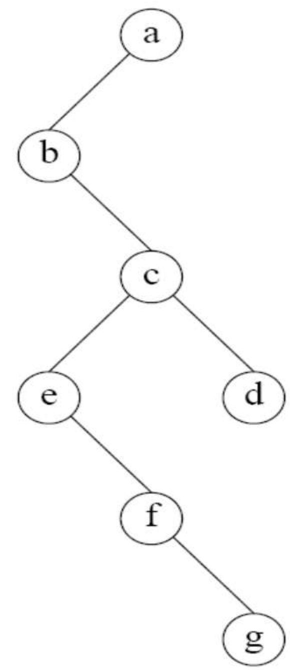
First-child, next-sibling trees

- A binary tree, used to represent n-ary (general) trees.
- Left child is first child.
- Right child is next sibling.

N-ary tree:



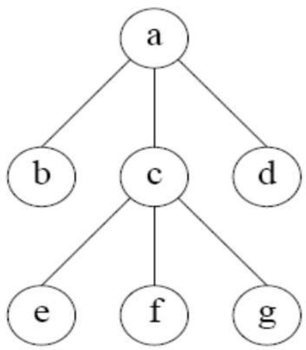
First-child,
next-sibling tree:

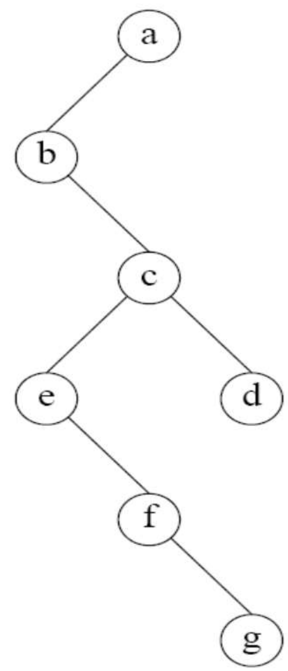


Advantage of first-child, next-sibling trees

- Pre-order traversal is the same.
  - Useful to print a tree, in indented format.

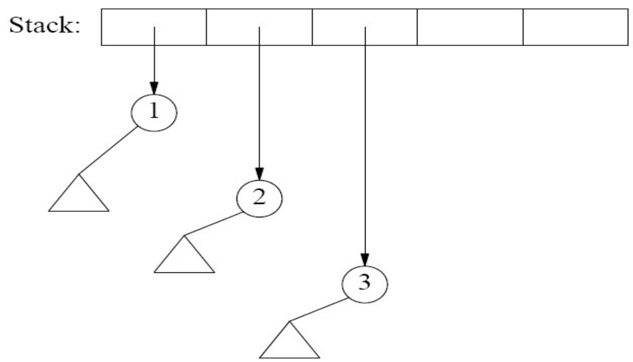
```
a(3)
... b(0)
... c(3)
...... e(0)
...... f(0)
...... g(0)
... d(0)
```



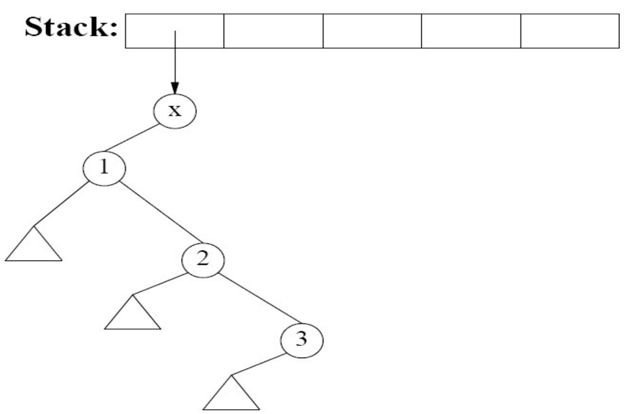


The build_tree procedure

Build_tree('x',3):



Result:




```
proc Build_tree(x,n);
  p=nil;
  for i=1 to n do
        c=pop(S);
        c.right=p;
        p=c;
  end;
  Push(S,node(x,p,nil));
end;
```

works with n=0, too.

summary

Red: done

- Possibilities:
  - Derivation tree or **Abstract Syntax Tree**.
  - Top-down, or **Bottom-up**.
  - For **original** or modified grammar !
- OUR FINAL GOAL ! (“the one”)
- Build the AST, for the original grammar, bottom-up.
- This is THE way to build a parser by hand.

Acknowledgements

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