Abstract Syntax Tree

- Condensed form of parse tree,
- useful for representing language constructs.
- The production $S \rightarrow \text{if } B \text{ then } s1 \text{ else } s2$
  may appear as

```
if
-
then
-
else

B

s1

s2
```
- Chain of single productions may be collapsed, and operators move to the parent nodes
Constructing Abstract Syntax Tree for expression

• Each node can be represented as a record
• *operators*: one field for operator, remaining fields ptrs to operands
  
  \[ \text{mknode}(\text{op}, \text{left}, \text{right}) \]

• *identifier*: one field with label id and another ptr to symbol table
  
  \[ \text{mkleaf}(\text{id}, \text{entry}) \]

• *number*: one field with label num and another to keep the value of the number
  
  \[ \text{mkleaf}(\text{num}, \text{val}) \]
Example

the following sequence of function calls creates a parse tree for a- 4 + c

\[ P_1 = \text{mkleaf}(\text{id, entry.a}) \]
\[ P_2 = \text{mkleaf}(\text{num, 4}) \]
\[ P_3 = \text{mknode}(-, P_1, P_2) \]
\[ P_4 = \text{mkleaf}(\text{id, entry.c}) \]
\[ P_5 = \text{mknode}(+, P_3, P_4) \]
A syntax directed definition for constructing syntax tree

\[
\begin{align*}
E & \rightarrow E_1 + T & E.\text{ptr} & = \text{mknode}(+, E_1.\text{ptr}, T.\text{ptr}) \\
E & \rightarrow T & E.\text{ptr} & = T.\text{ptr} \\
T & \rightarrow T_1 * F & T.\text{ptr} & := \text{mknode}(*, T_1.\text{ptr}, F.\text{ptr}) \\
T & \rightarrow F & T.\text{ptr} & := F.\text{ptr} \\
F & \rightarrow (E) & F.\text{ptr} & := E.\text{ptr} \\
F & \rightarrow id & F.\text{ptr} & := \text{mkleaf}(id, \text{entry.id}) \\
F & \rightarrow \text{num} & F.\text{ptr} & := \text{mkleaf}(\text{num}, \text{val})
\end{align*}
\]
DAG for Expressions

Expression $a + a * (b - c) + (b - c) * d$
make a leaf or node if not present, otherwise return pointer to the existing node

$P_1 = \text{makeleaf}(id, a)$
$P_2 = \text{makeleaf}(id, a)$
$P_3 = \text{makeleaf}(id, b)$
$P_4 = \text{makeleaf}(id, c)$
$P_5 = \text{makenode}(-, P_3, P_4)$
$P_6 = \text{makenode}(*, P_2, P_5)$
$P_7 = \text{makenode}(+, P_1, P_6)$
$P_8 = \text{makeleaf}(id, b)$
$P_9 = \text{makeleaf}(id, c)$
$P_{10} = \text{makenode}(-, P_8, P_9)$
$P_{11} = \text{makeleaf}(id, d)$
$P_{12} = \text{makenode}(*, P_{10}, P_{11})$
$P_{13} = \text{makenode}(+, P_7, P_{12})$
Bottom-up evaluation of S-attributed definitions

• Can be evaluated while parsing
• Whenever reduction is made, value of new synthesized attribute is computed from the attributes on the stack
• Extend stack to hold the values also
• The current top of stack is indicated by top pointer
Suppose semantic rule
\[ A.a = f(X.x, Y.y, Z.z) \]
is associated with production
\[ A \rightarrow XYZ \]
Before reducing \( XYZ \) to \( A \), value of \( Z \) is in \( \text{val(top)} \), value of \( Y \) is in \( \text{val(top-1)} \) and value of \( X \) is in \( \text{val(top-2)} \)
If symbol has no attribute then the entry is undefined
After the reduction, \( \text{top} \) is decremented by 2 and state covering \( A \) is put in \( \text{val(top)} \)
Example: desk calculator

L → E $  
E → E + T  
E → T  
T → T * F  
T → F  
F → (E)  
F → digit  

Print (E.val)  
E.val = E.val + T.val  
E.val = T.val  
T.val = T.val * F.val  
T.val = F.val  
F.val = E.val  
F.val = digit.lexval
**Example:** desk calculator

<table>
<thead>
<tr>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>L → E$</td>
</tr>
<tr>
<td>E → E + T</td>
</tr>
<tr>
<td>E → T</td>
</tr>
<tr>
<td>T → T * F</td>
</tr>
<tr>
<td>T → F</td>
</tr>
<tr>
<td>F → (E)</td>
</tr>
<tr>
<td>F → digit</td>
</tr>
</tbody>
</table>

Before reduction \( ntop = top - r + 1 \)

After code reduction \( top = ntop \)

\( r \) is the #symbols on RHS
<table>
<thead>
<tr>
<th>INPUT</th>
<th>STATE</th>
<th>Val</th>
<th>PROD</th>
</tr>
</thead>
<tbody>
<tr>
<td>3*5+4$</td>
<td>digit</td>
<td>3</td>
<td>F → digit</td>
</tr>
<tr>
<td>*5+4$</td>
<td>F</td>
<td>3</td>
<td>T → F</td>
</tr>
<tr>
<td>*5+4$</td>
<td>T</td>
<td>3</td>
<td>T*</td>
</tr>
<tr>
<td>*5+4$</td>
<td>T*</td>
<td>3</td>
<td>T* → digit</td>
</tr>
<tr>
<td>5+4$</td>
<td>T*</td>
<td>3</td>
<td>T* → F</td>
</tr>
<tr>
<td>+4$</td>
<td>T</td>
<td>15</td>
<td>T → T * F</td>
</tr>
<tr>
<td>+4$</td>
<td>E</td>
<td>15</td>
<td>E → T</td>
</tr>
<tr>
<td>+4$</td>
<td>E</td>
<td>15</td>
<td>E+</td>
</tr>
<tr>
<td>+4$</td>
<td>E+</td>
<td>15</td>
<td>E+ → digit</td>
</tr>
<tr>
<td>4$</td>
<td>E+</td>
<td>15</td>
<td>E+ → F</td>
</tr>
<tr>
<td>$</td>
<td>E+</td>
<td>15</td>
<td>E+ → T</td>
</tr>
<tr>
<td>$</td>
<td>E+T</td>
<td>15</td>
<td>E+ → E +T</td>
</tr>
<tr>
<td>$</td>
<td>E</td>
<td>19</td>
<td>E → E + T</td>
</tr>
<tr>
<td>$</td>
<td></td>
<td></td>
<td>E → E + T</td>
</tr>
<tr>
<td>$</td>
<td></td>
<td></td>
<td>E → E + T</td>
</tr>
</tbody>
</table>

12
YACC Terminology

\[ E \rightarrow E + T \]  \[ \text{val}(\text{ntop}) = \text{val}(\text{top}-2) + \text{val}(\text{top}) \]

In YACC
\[ E \rightarrow E + T \]  \[ $$ = $1 + $3 \]

$$ maps to \text{val}[\text{top} - r + 1]
$k$ maps to \text{val}[\text{top} - r + k]
r = \#\text{symbols on RHS (here 3)}
$$ = $1$ is the default action in YACC
L-attributed definitions

• When translation takes place during parsing, order of evaluation is linked to the order in which nodes are created.
• In S-attributed definitions parent’s attribute evaluated after child’s.
• A natural order in both top-down and bottom-up parsing is depth-first order.
• **L-attributed** definition: where attributes can be evaluated in depth-first order.
L attributed definitions ...

- A syntax directed definition is L-attributed if each inherited attribute of $X_j$ (1 ≤ j ≤ n) at the right hand side of $A \rightarrow X_1 X_2...X_n$ depends only on
  - Attributes of symbols $X_1 X_2...X_{j-1}$ and
  - Inherited attribute of $A$

- Examples (i inherited, s synthesized)

  \[
  A \rightarrow LM \quad L.i = f_1(A.i) \\
  M.i = f_2(L.s) \\
  A.s = f_3(M.s)
  \]

  \[
  A \rightarrow QR \quad R.i = f_4(A.i) \\
  Q.i = f_5(R.s) \\
  A.s = f_6(Q.s)
  \]
Translation schemes

• A CFG where semantic actions occur within the rhs of production

• Example: A translation scheme to map infix to postfix
  \[ E \rightarrow T \ R \]
  \[ R \rightarrow \text{addop} \ T \ R \mid \varepsilon \]
  \[ T \rightarrow \text{num} \]
  \[ \text{addop} \rightarrow + \mid – \]

Exercise: Create Parse Tree for 9 – 5 + 2
Evaluation of Translation Schemes

- Assume actions are terminal symbols
- Perform depth first order traversal to obtain $9 5 - 2 +$

- When designing translation scheme, ensure attribute value is available when referred to
- In case of synthesized attribute it is trivial (why?)
• An inherited attribute for a symbol on RHS of a production must be computed in an action before that symbol
S → A₁ A₂ \{A₁.\text{in} = 1, A₂.\text{in} = 2\}
A → a \{\text{print}(A.\text{in})\}

depth first order traversal gives error \textbf{(undef)}

• A synthesized attribute for the non terminal on the LHS can be computed after all attributes it references, have been computed. \textbf{The action normally should be placed at the end of RHS.}
Bottom up evaluation of inherited attributes

• Remove embedded actions from translation scheme
• Make transformation so that embedded actions occur only at the ends of their productions
• Replace each action by a distinct marker non terminal M and attach action at end of M → ε
E → T R
R → + T \{\text{print (}) \}\ R
R → - T \{\text{print (}}\ R
R → \epsilon
T → \text{num} \ \{\text{print(num.val)}\}

transforms to

E → T R
R → + T \ M \ R
R → - T \ N \ R
R → \epsilon
T → \text{num} \ \{\text{print(num.val)}\}
M → \epsilon \ \{\text{print(}+\})\}
N → \epsilon \ \{\text{print(-)}\}
Inheriting attribute on parser stacks

- bottom up parser reduces rhs of $A \rightarrow XY$ by removing $XY$ from stack and putting $A$ on the stack

- synthesized attributes of $X$s can be inherited by $Y$ by using the copy rule $Y.i=X.s$
Inherited Attributes: SDD

D → T L

L.in = T.type

T → real

T.type = real

T int

T.type = int

L → L₁, id

L₁.in = L.in;
addtype(id.entry, L.in)

L → id

addtype (id.entry,L.in)

Exercise: Convert to Translation Scheme
Inherited Attributes: Translation Scheme

D $\rightarrow$ T \{L.in = T.type\} L

T $\rightarrow$ int \{T.type = integer\}
T $\rightarrow$ real \{T.type = real\}

L $\rightarrow$ \{L_1.in = L.in\} L_1.id \{addtype(id.entry,L_{in})\}

L $\rightarrow$ id \{addtype(id.entry,L_{in})\}

Example: take string real p,q,r
### State stack

<table>
<thead>
<tr>
<th>State stack</th>
<th>INPUT</th>
<th>PRODUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>real</td>
<td>real p,q,r</td>
<td>real p,q,r</td>
</tr>
<tr>
<td>T</td>
<td>p,q,r</td>
<td>T → real</td>
</tr>
<tr>
<td>Tp</td>
<td>,q,r</td>
<td></td>
</tr>
<tr>
<td>TL</td>
<td>,q,r</td>
<td>L → id</td>
</tr>
<tr>
<td>TL,</td>
<td>q,r</td>
<td></td>
</tr>
<tr>
<td>TL,q</td>
<td>,r</td>
<td></td>
</tr>
<tr>
<td>TL</td>
<td>,r</td>
<td>L → L,id</td>
</tr>
<tr>
<td>TL,</td>
<td>r</td>
<td></td>
</tr>
<tr>
<td>TL,r</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>TL</td>
<td>-</td>
<td>L → L,id</td>
</tr>
<tr>
<td>D</td>
<td>-</td>
<td>D → TL</td>
</tr>
</tbody>
</table>

Every time a string is reduced to L, T.val is just below it on the stack.
Example ...

- Every time a reduction to L is made value of T type is just below it.
- Use the fact that T.val (type information) is at a known place in the stack.
- When production $L \rightarrow id$ is applied, id.entry is at the top of the stack and T.type is just below it, therefore,
  \[
  \text{addtype}(id.entry, L.in) \iff \text{addtype}(\text{val}[\text{top}], \text{val}[\text{top}-1])
  \]
- Similarly when production $L \rightarrow L_1$, id is applied id.entry is at the top of the stack and T.type is three places below it, therefore,
  \[
  \text{addtype}(id.entry, L.in) \iff \text{addtype}(\text{val}[\text{top}], \text{val}[\text{top}-3])
  \]
Example ...

Therefore, the translation scheme becomes

\[ D \rightarrow T \ L \]
\[ T \rightarrow \text{int} \quad \text{val}[\text{top}] = \text{integer} \]
\[ T \rightarrow \text{real} \quad \text{val}[\text{top}] = \text{real} \]

\[ L \rightarrow L, \text{id} \quad \text{addtype(\text{val}[\text{top}], \text{val}[\text{top}-3])} \]
\[ L \rightarrow \text{id} \quad \text{addtype(\text{val}[\text{top}], \text{val}[\text{top}-1])} \]
Simulating the evaluation of inherited attributes

• The scheme works only if grammar allows position of attribute to be predicted.
• Consider the grammar
  
  \[ S \rightarrow aAC \quad C_i = A_s \]
  
  \[ S \rightarrow bABC \quad C_i = A_s \]
  
  \[ C \rightarrow c \quad C_s = g(C_i) \]

• C inherits \( A_s \)
• there may or may not be a B between A and C on the stack when reduction by rule \( C \rightarrow c \) takes place
• When reduction by \( C \rightarrow c \) is performed the value of \( C_i \) is either in [top-1] or [top-2]
Simulating the evaluation ...

• Insert a marker $M$ just before $C$ in the second rule and change rules to

\[
\begin{align*}
S & \rightarrow aAC & C_i &= A_s \\
S & \rightarrow bABMC & M_i &= A_s \ ; \ C_i &= M_s \\
C & \rightarrow c & C_s &= g(C_i) \\
M & \rightarrow \epsilon & M_s &= M_i
\end{align*}
\]

• When production $M \rightarrow \epsilon$ is applied we have $M_s = M_i = A_s$

• Therefore value of $C_i$ is always at val[top-1]
Simulating the evaluation ...

- Markers can also be used to simulate rules that are not copy rules

\[
S \rightarrow aAC \quad \quad C_i = f(A.s)
\]

- using a marker

\[
\begin{align*}
S & \rightarrow aANC \\
N & \rightarrow \varepsilon \\
N & \rightarrow \varepsilon
\end{align*}
\]

\[
\begin{align*}
N_i &= A_s; \quad C_i = N_s \\
N_s &= f(N_i)
\end{align*}
\]
General algorithm

- **Algorithm**: Bottom up parsing and translation with inherited attributes
- **Input**: L attributed definitions
- **Output**: A bottom up parser

- Assume every non terminal has one inherited attribute and every grammar symbol has a synthesized attribute

- For every production $A \rightarrow X_1 \ldots X_n$ introduce $n$ markers $M_1 \ldots M_n$ and replace the production by
  
  $A \rightarrow M_1 X_1 \ldots M_n X_n$
  
  $M_1 \ldots M_n \rightarrow \epsilon$

- Synthesized attribute $X_{j,s}$ goes into the value entry of $X_j$

- Inherited attribute $X_{j,i}$ goes into the value entry of $M_j$
Algorithm ...

• If the reduction is to a marker $M_j$ and the marker belongs to a production $A \rightarrow M_1 X_1... M_n X_n$ then

$A$ is in position top-$2j+2$
$X_1$ is in position top-$2j+3$
$X_1.s$ is in position top-$2j+4$

• If reduction is to a non terminal $A$ by production $A \rightarrow M_1 X_1... M_n X_n$ then compute $A_s$ and push on the stack
Space for attributes at compile time

• Lifetime of an attribute begins when it is first computed

• Lifetime of an attribute ends when all the attributes depending on it, have been computed

• Space can be conserved by assigning space for an attribute only during its lifetime
Example

- Consider following definition

\[ D \rightarrow T \, L \quad L.in := T.type \]
\[ T \rightarrow \text{real} \quad T.type := \text{real} \]
\[ T \rightarrow \text{int} \quad T.type := \text{int} \]
\[ L \rightarrow L_1, l \quad L_1.in := L.in; \, l.in=L.in \]
\[ L \rightarrow l \quad l.in = L.in \]
\[ l \rightarrow l_1[\text{num}] \quad l_1.in=\text{array}(\text{numeral}, \, l.in) \]
\[ l \rightarrow \text{id} \quad \text{addtype}(\text{id}.entry, l.in) \]
Consider string \texttt{int x[3], y[5]}

its parse tree and dependence graph
Allocate resources using life time information

R1  R1  R2  R3  R2  R1  R1  R2  R1

Allocate resources using life time and copy information

R1  =R1  =R1  R2  R2  =R1  =R1  R2  R1
Space for attributes at compiler Construction time

• Attributes can be held on a single stack. However, lot of attributes are copies of other attributes.

• For a rule like $A \rightarrow B \ C$ stack grows up to a height of five (assuming each symbol has one inherited and one synthesized attribute).

• Just before reduction by the rule $A \rightarrow B \ C$ the stack contains $I(A) \ I(B) \ S(B) \ I(C) \ S(C)$.

• After reduction the stack contains $I(A) \ S(A)$.
Example

• Consider rule $B \rightarrow B_1 B_2$ with inherited attribute $ps$ and synthesized attribute $ht$

• The parse tree for this string and a snapshot of the stack at each node appears as
Example ...

- However, if different stacks are maintained for the inherited and synthesized attributes, the stacks will normally be smaller.