Lecture 05 – Syntax analysis & Semantic Analysis

THEORY OF COMPILATION

Eran Yahav
Last week:
LR Parsing with Pushdown Automaton

input

state

symbol

output

stack

\[
\begin{array}{c}
 q_0 \\
 i \\
 q_5 \\
\end{array}
\]

top (current state)


ACTION Table

GOTO Table

\[ \text{lookahead} \]

\[ \text{state} \]

\[ \text{state} \]
Last week:
LR Parsing with Pushdown Automaton

- $s =$ top of stack, $t =$ next token, use $\text{ACTION}[s][t]$ to determine what is the next move

- **Shift move**
  - Remove first token $t$ from input
  - Push $t$ on the stack
  - Compute next state $s' = \text{GOTO}[s][t]$ table
  - Push new state $s'$ on the stack
  - If new state is error – report error

- **Reduce move**
  - Using a rule $N \rightarrow \alpha$
  - Symbols in $\alpha$ and their following states are removed from stack. Let $q$ denote the state on top of stack after their removal
  - Push $N$ on the stack
  - Compute next state $s' = \text{GOTO}[q][N]$ table
  - Push new state $s'$ on the stack (on top of $N$)
Last week: shift move

1. Remove first token t from input
2. Push t on the stack
3. Compute s’ = GOTO[s][t] table
4. Push s’ state on the stack
5. If new state is error – report error
Last week: reduce move

1. Using a rule $N \rightarrow \alpha$ (ACTION[s][t])
2. Symbols in $\alpha$ and their following states are removed from stack. $q = \text{top afterwards}$.
3. Push $N$ on the stack
4. New state $s' = \text{GOTO}[q][N]$ table
5. Push new state $s'$ on top of $N
Constructing Parse Table: LR(0) Automaton Example
### Last week: GOTO/ACTION Table

<table>
<thead>
<tr>
<th>State</th>
<th>i</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
<th>E</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>q0</td>
<td>s5</td>
<td></td>
<td>s7</td>
<td></td>
<td></td>
<td>s1</td>
<td>s6</td>
</tr>
<tr>
<td>q1</td>
<td></td>
<td>s3</td>
<td></td>
<td></td>
<td></td>
<td>s2</td>
<td></td>
</tr>
<tr>
<td>q2</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
</tr>
<tr>
<td>q3</td>
<td>s5</td>
<td></td>
<td>s7</td>
<td></td>
<td></td>
<td></td>
<td>s4</td>
</tr>
<tr>
<td>q4</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
</tr>
<tr>
<td>q5</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
</tr>
<tr>
<td>q6</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
</tr>
<tr>
<td>q7</td>
<td>s5</td>
<td></td>
<td>s7</td>
<td></td>
<td></td>
<td></td>
<td>s8</td>
</tr>
<tr>
<td>q8</td>
<td></td>
<td>s3</td>
<td></td>
<td></td>
<td></td>
<td>s9</td>
<td></td>
</tr>
<tr>
<td>q9</td>
<td>r5</td>
<td>r5</td>
<td>r5</td>
<td>r5</td>
<td>r5</td>
<td>r5</td>
<td>r5</td>
</tr>
</tbody>
</table>

(1) $Z \rightarrow E \; \$ 
(2) $E \rightarrow T$ 
(3) $E \rightarrow E \; + \; T$ 
(4) $T \rightarrow i$ 
(5) $T \rightarrow (\; E \; )$

Warning: numbers mean different things!

rn = reduce using rule number n 
sm = shift to state m
LR Parsing with Pushdown Automaton (superimposed GOTO/ACTION)

- $s =$ top of stack, $t =$ next token,
  move = GOTO/ACTION[$s$][$t$] to determine what is the next move

- If (move = $Sm$)
  - Remove first token $t$ from input
  - Push $t$ on the stack
  - Push new state $m$ on the stack

- If (move = $rn$)
  - use rule number $n$: $N \rightarrow \alpha$
  - Symbols in $\alpha$ and their following states are removed from stack. Let $q$ denote the state on top of stack after their removal
  - Push $N$ on the stack
  - Compute next state $s' =$ GOTO/ACTION[$q$][$N$] table
  - Push new state $s'$ on the stack (on top of $N$)

- If (move = empty) report ERROR
GOTO/ACTION Table

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>q0</td>
<td>i + i $</td>
<td>s5</td>
</tr>
<tr>
<td>q0 i q5</td>
<td>+ i $</td>
<td>r4</td>
</tr>
<tr>
<td>q0 T q6</td>
<td>+ i $</td>
<td>r2</td>
</tr>
<tr>
<td>q0 E q1</td>
<td>+ i $</td>
<td>s3</td>
</tr>
<tr>
<td>q0 E q1 + q3</td>
<td>i $</td>
<td>s5</td>
</tr>
<tr>
<td>q0 E q1 + q3 i q5</td>
<td>$</td>
<td>r4</td>
</tr>
<tr>
<td>q0 E q1 + q3 T q4</td>
<td>$</td>
<td>r3</td>
</tr>
<tr>
<td>q0 E q1</td>
<td>$</td>
<td>s2</td>
</tr>
<tr>
<td>q0 E q1 $ q2</td>
<td></td>
<td>r1</td>
</tr>
<tr>
<td>q0 Z</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(1) Z → E $  
(2) E → T  
(3) E → E + T  
(4) T → i  
(5) T → ( E )
Are we done?

- Can make a transition diagram for any grammar
- Can make a GOTO table for every grammar
- Cannot make a deterministic ACTION table for every grammar
LR(0) Conflicts

Z → •E$
E → •T
E → •E + T
T → •i
T → •(E)
T → •i[E]

Shift/reduce conflict

Z → E $
E → T
E → E + T
T → i
T → ( E )
T → i[E]
LR(0) Conflicts

Z → •E$
E → •T
E → •E + T
T → •i
T → •(E)
V → •i

reduce/reduce conflict

Z → E $
E → T
E → E + T
T → i
V → i
T → ( E )
LR(0) Conflicts

- Any grammar with an $\varepsilon$-rule cannot be LR(0)
- Inherent shift/reduce conflict
  - $A \rightarrow \varepsilon \bullet$ - reduce item
  - $P \rightarrow \alpha \bullet A \beta$ – shift item
  - $A \rightarrow \varepsilon \bullet$ can always be predicted from $P \rightarrow \alpha \bullet A \beta$
Back to the GOTO/ ACTIONS tables

<table>
<thead>
<tr>
<th>State</th>
<th>i</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
<th>E</th>
<th>T</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>q0</td>
<td>q5</td>
<td>q7</td>
<td></td>
<td></td>
<td></td>
<td>q1</td>
<td>q6</td>
<td>shift</td>
</tr>
<tr>
<td>q1</td>
<td></td>
<td>q3</td>
<td></td>
<td></td>
<td></td>
<td>q2</td>
<td></td>
<td>shift</td>
</tr>
<tr>
<td>q2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Z$E</td>
</tr>
<tr>
<td>q3</td>
<td>q5</td>
<td>q7</td>
<td></td>
<td></td>
<td></td>
<td>q4</td>
<td></td>
<td>Shift</td>
</tr>
<tr>
<td>q4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E$E+T</td>
</tr>
<tr>
<td>q5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T$i</td>
</tr>
<tr>
<td>q6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E$T</td>
</tr>
<tr>
<td>q7</td>
<td>q5</td>
<td>q7</td>
<td></td>
<td></td>
<td></td>
<td>q8</td>
<td>q6</td>
<td>shift</td>
</tr>
<tr>
<td>q8</td>
<td></td>
<td>q3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>shift</td>
</tr>
<tr>
<td>q9</td>
<td></td>
<td>q3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T$E</td>
</tr>
</tbody>
</table>

ACTION table determined only by transition diagram, ignores input
SLR Grammars

- Don’t reduce if it will get you into trouble on the next token
- A handle should not be reduced to a non-terminal N if the look-ahead is a token that cannot follow N
- A reduce item N → α● is applicable only when the look-ahead is in FOLLOW(N)
- Differs from LR(0) only on the ACTION table
LR(0) Conflicts

Shift/reduce conflict

Z → •E$
E → •T
E → •E + T
T → •i
T → •(E)
T → •i[E]

FOLLOW(Z) = { $ }
FOLLOW(E) = { ) + $ }
FOLLOW(T) = { ) + $ }

A[x]$

input

q₀

q₅

...
### SLR ACTION Table

<table>
<thead>
<tr>
<th>State</th>
<th>i</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>q0</td>
<td>shift</td>
<td>shift</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q1</td>
<td>shift</td>
<td></td>
<td>shift</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q2</td>
<td></td>
<td></td>
<td>Z→E$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q3</td>
<td>shift</td>
<td></td>
<td>shift</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q4</td>
<td>E→E+T</td>
<td>E→E+T</td>
<td>E→E+T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q5</td>
<td>T→i</td>
<td>T→i</td>
<td>T→i</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q6</td>
<td>E→T</td>
<td>E→T</td>
<td>E→T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q7</td>
<td>shift</td>
<td></td>
<td></td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>q8</td>
<td>shift</td>
<td></td>
<td></td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>q9</td>
<td>T→(E)</td>
<td>T→(E)</td>
<td>T→(E)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(1) $Z \rightarrow E \; \$ $
(2) $E \rightarrow T$
(3) $E \rightarrow E \; + \; T$
(4) $T \rightarrow i$
(5) $T \rightarrow ( \; E \; )$

**Look-ahead token from the input**

**Remember:** 
**In contrast,** GOTO table is indexed by state and a grammar symbol from the stack

FOLLOW($Z$) = { $ $ }
FOLLOW($E$) = { ) + $ $ }
FOLLOW($T$) = { ) + $ $ }
### SLR ACTION Table

<table>
<thead>
<tr>
<th>State</th>
<th>i</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>[</th>
<th>]</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>q0</td>
<td>shift</td>
<td>shift</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q1</td>
<td>shift</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>shift</td>
</tr>
<tr>
<td>q2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Z→E$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q3</td>
<td>shift</td>
<td>shift</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q4</td>
<td>E→E+T</td>
<td>E→E+T</td>
<td>E→E+T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q5</td>
<td>T→i</td>
<td>T→i</td>
<td>shift</td>
<td>T→i</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q6</td>
<td>E→T</td>
<td>E→T</td>
<td>E→T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q7</td>
<td>shift</td>
<td>shift</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q8</td>
<td>shift</td>
<td>shift</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q9</td>
<td>T→(E)</td>
<td>T→(E)</td>
<td>T→(E)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SLR – use 1 token look-ahead**

... as before...

T \to i

T \to i[E]

FOLLOW(Z) = \{ $ \}

FOLLOW(E) = \{ ) + $ \}

FOLLOW(T) = \{ ) + $ \}

### vs.

<table>
<thead>
<tr>
<th>state</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>q0</td>
<td>shift</td>
</tr>
<tr>
<td>q1</td>
<td>shift</td>
</tr>
<tr>
<td>q2</td>
<td>Z→E$</td>
</tr>
<tr>
<td>q3</td>
<td>Shift</td>
</tr>
<tr>
<td>q4</td>
<td>E→E+T</td>
</tr>
<tr>
<td>q5</td>
<td>T→i</td>
</tr>
<tr>
<td>q6</td>
<td>E→T</td>
</tr>
<tr>
<td>q7</td>
<td>shift</td>
</tr>
<tr>
<td>q8</td>
<td>shift</td>
</tr>
<tr>
<td>q9</td>
<td>T→E</td>
</tr>
</tbody>
</table>

**LR(0) – no look-ahead**

---

19
Are we done?

(0) $S' \rightarrow S$
(1) $S \rightarrow L = R$
(2) $S \rightarrow R$
(3) $L \rightarrow * R$
(4) $L \rightarrow id$
(5) $R \rightarrow L$
Shift/reduce conflict

- $S \rightarrow L \bullet = R$ vs. $R \rightarrow L \bullet$
- FOLLOW($R$) contains $=$
  - $S \Rightarrow L = R \Rightarrow * R = R$
- SLR cannot resolve the conflict either
LR(1) Grammars

- In SLR: a reduce item $N \rightarrow \alpha \cdot$ is applicable only when the look-ahead is in FOLLOW($N$).
- But FOLLOW($N$) merges look-ahead for all alternatives for $N$.

- LR(1) keeps look-ahead with each LR item.

- Idea: a more refined notion of follows computed per item.
LR(1) Item

- LR(1) item is a pair
  - LR(0) item
  - Look-ahead token

- Meaning
  - We matched the part left of the dot, looking to match the part on the right of the dot, followed by the look-ahead token.

- Example
  - The production $L \rightarrow id$ yields the following LR(1) items

\[
\begin{align*}
(0) & \quad S' \rightarrow S \\
(1) & \quad S \rightarrow L = R \\
(2) & \quad S \rightarrow R \\
(3) & \quad L \rightarrow * R \\
(4) & \quad L \rightarrow id \\
(5) & \quad R \rightarrow L
\end{align*}
\]

\[
\begin{align*}
[L \rightarrow \bullet id, \star] \\
[L \rightarrow \bullet id, =] \\
[L \rightarrow \bullet id, id] \\
[L \rightarrow \bullet id, \$] \\
[L \rightarrow id \bullet, \star] \\
[L \rightarrow id \bullet, =] \\
[L \rightarrow id \bullet, id] \\
[L \rightarrow id \bullet, \$]
\end{align*}
\]
\( \varepsilon \)-closure for LR(1)

- For every \([A \rightarrow \alpha \bullet B\beta , c]\) in \(S\)
  - for every production \(B \rightarrow \delta\) and every token \(b\) in the grammar such that \(b \in \text{FIRST}(\beta c)\)
  - Add \([B \rightarrow \bullet \delta , b]\) to \(S\)
Back to the conflict

- Is there a conflict now?
LALR

- LR tables have large number of entries
- Often don’t need such refined observation (and cost)
- LALR idea: find states with the same LR(0) component and merge their look-ahead component as long as there are no conflicts
- LALR not as powerful as LR(1)
Summary: LR Grammars

- LR parsing techniques use item sets of proposed handles
  - Shift behavior similar
  - Differ on when to reduce

- LR(0) - any reduce item causes a reduction
- SLR – a reduce item \( N \rightarrow \alpha \bullet \) causes a reduction only if the look-ahead token is in the FOLLOW set of \( N \)
- LR(1) - a reduce item \( N \rightarrow \alpha \bullet \{\sigma\} \) causes a reduction only if the look-ahead token is in the set \( \sigma \) (the look-ahead set computed for the item)
Summary: LR Grammars

- ACTION table determines whether to shift or reduce
- On a shift, new state found using the GOTO table
- LR-parser with 1 token look-ahead, the ACTION and GOTO tables can be superimposed
Summary

- Bottom up
  - LR Items
  - LR parsing with pushdown automata
  - LR(0), SLR, LR(1) – different kinds of LR items, same basic algorithm
You are here...

Source text

Process text input

Lexical Analysis tokens

Syntax Analysis AST

Sem. Analysis

Annotated AST

Back End

Intermediate code generation

Intermediate code optimization IR

Code generation

Target code optimization

Symbolic Instructions SI

Machine code generation

Write executable output MI

Executable code

exe
What we want

Potato potato;
Carrot carrot;
\( x = \text{tomato} + \text{potato} + \text{carrot} \)

Lexical analyzer

\(<\text{id,tomato}>,<\text{PLUS}>,<\text{id,potato}>,<\text{PLUS}>,<\text{id,carrot}>,\text{EOF}\)

Parser

<table>
<thead>
<tr>
<th>symbol</th>
<th>kind</th>
<th>type</th>
<th>properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>var</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>tomato</td>
<td>var</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>potato</td>
<td>var</td>
<td>Potato</td>
<td></td>
</tr>
<tr>
<td>carrot</td>
<td>var</td>
<td>Carrot</td>
<td></td>
</tr>
</tbody>
</table>

tomato is undefined
potato used before initialized
Cannot add Potato and Carrot
Syntax vs. Semantics

- Syntax
  - Program structure
  - Formally described via context free grammars

- Semantics
  - Program meaning
  - Formally defined as various forms of semantics (e.g., operational, denotational)
  - It is actually NOT what “semantic analysis” phase does
  - Better name – “contextual analysis”
Contextual Analysis

- Often called “Semantic analysis”

- Properties that cannot be formulated via CFG
  - Type checking
  - Declare before use
    - Identifying the same word “w” re-appearing – wbw
  - Initialization
  - ...

- Properties that are hard to formulate via CFG
  - “break” only appears inside a loop
  - ...

- Processing of the AST
Abstract Syntax Tree (AST)

- Abstract away some syntactic details of the source language

\[
S \rightarrow \text{if } E \text{ then } S \text{ else } S \\
| \quad \ldots
\]

\[
\begin{align*}
\text{if } (x>0) \\
\text{then } & \{ y = 42 \} \\
\text{else } & \{ y = 73 \}
\end{align*}
\]
Parse tree (concrete syntax tree)

```
S
|-- if
|   |-- E
|       |-- (x)
|            |-- E
|                |-- id
|                     |-- >
|                          |-- id
|                                           |-- o

S
|-- then
|   |-- E
|       |-- {S}
|            |-- id
|                |-- =
|                    |-- num

S
|-- else
|   |-- S
|       |-- {S}
|            |-- id
|                |-- =
|                    |-- num
```
Abstract Syntax Tree (AST)
Syntax Directed Translation

- Semantic attributes
  - Attributes attached to grammar symbols
- Semantic actions
  - (already mentioned when we did recursive descent)
  - How to update the attributes
- Attribute grammars
Attribute grammars

- Attributes
  - Every grammar symbol has attached attributes
    - Example: Expr.type

- Semantic actions
  - Every production rule can define how to assign values to attributes
    - Example:
      \[\text{Expr} \rightarrow \text{Expr} + \text{Term}\]
      \[\text{Expr.type} = \text{Expr1.type} \text{ when } (\text{Expr1.type} == \text{Term.type})\]
      \[\text{Error otherwise}\]
Indexed symbols

- Add indexes to distinguish repeated grammar symbols
- Does not affect grammar
- Used in semantic actions

- $\text{Expr} \rightarrow \text{Expr} + \text{Term}$
  Becomes
- $\text{Expr} \rightarrow \text{Expr}_1 + \text{Term}$
Example

Production | Semantic Rule  
---|---
D → TL | L.in = T.type  
T → int | T.type = integer  
T → float | T.type = float  
L → L1, id | L1.in = L.in  
  | addType(id.entry,L.in)  
L → id | addType(id.entry,L.in)  

```text
float x,y,z
```
Dependencies

- A semantic equation $a = b_1, \ldots, b_m$ requires computation of $b_1, \ldots, b_m$ to determine the value of $a$

- The value of $a$ depends on $b_1, \ldots, b_m$
  - We write $a \leftarrow b_i$
Attribute Evaluation

- Build the AST
- Fill attributes of terminals with values derived from their representation
- Execute evaluation rules of the nodes to assign values until no new values can be assigned
  - In the right order such that
    - No attribute value is used before its available
    - Each attribute will get a value only once
Cycles

- Cycle in the dependence graph
- May not be able to compute attribute values

E.S = T.i
T.i = E.s + 1
Attribute Evaluation

- Build the AST
- Build dependency graph
- Compute evaluation order using topological ordering
- Execute evaluation rules based on topological ordering

- Works as long as there are no cycles
Building Dependency Graph

- All semantic equations take the form
  
  $\text{attr}_1 = \text{func}_1(\text{attr}_1.1, \text{attr}_1.2, \ldots)$
  $\text{attr}_2 = \text{func}_2(\text{attr}_2.1, \text{attr}_2.2, \ldots)$

- Actions with side effects use a dummy attribute
- Build a directed dependency graph $G$
  - For every attribute $a$ of a node $n$ in the AST create a node $n.a$
  - For every node $n$ in the AST and a semantic action of the form $b = f(c_1, c_2, \ldots c_k)$ add edges of the form $(c_i, b)$
Example

```
float x, y, z

Prod. | Semantic Rule
--- | ---
D → T L | L.in = T.type
T → int | T.type = integer
T → float | T.type = float
L → L₁, id | L₁.in = L.in
            | addType(id.entry, L.in)
L → id | addType(id.entry, L.in)
```
**Example**

```
float x, y, z
```

<table>
<thead>
<tr>
<th>Prod.</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D \rightarrow TL$</td>
<td>$L.in = T.type$</td>
</tr>
<tr>
<td>$T \rightarrow int$</td>
<td>$T.type = integer$</td>
</tr>
<tr>
<td>$T \rightarrow float$</td>
<td>$T.type = float$</td>
</tr>
<tr>
<td>$L \rightarrow L_1, id$</td>
<td>$L_1.in = L.in$ addType(id.entry, L.in)</td>
</tr>
<tr>
<td>$L \rightarrow id$</td>
<td>addType(id.entry, L.in)</td>
</tr>
</tbody>
</table>
Topological Order

- For a graph $G=(V,E)$, $|V|=k$

- Ordering of the nodes $v_1,v_2,...v_k$ such that for every edge $(v_i,v_j) \in E$, $i < j$

Example topological orderings: 1 4 3 2 5, 4 1 3 5 2
Example

float x, y, z
But what about cycles?

- For a given attribute grammar hard to detect if it has cyclic dependencies
  - Exponential cost

- Special classes of attribute grammars
  - Our “usual trick”
  - sacrifice generality for predictable performance
Inherited vs. Synthesized Attributes

- **Synthesized attributes**
  - Computed from children of a node

- **Inherited attributes**
  - Computed from parents and siblings of a node

- Attributes of tokens are technically considered as synthesized attributes
example

float x, y, z

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>D → T L</td>
<td>L.in = T.type</td>
</tr>
<tr>
<td>T → int</td>
<td>T.type = integer</td>
</tr>
<tr>
<td>T → float</td>
<td>T.type = float</td>
</tr>
<tr>
<td>L → L₁, id</td>
<td>L₁.in = L.in</td>
</tr>
<tr>
<td></td>
<td>addType(id.entry, L.in)</td>
</tr>
<tr>
<td>L → id</td>
<td>addType(id.entry, L.in)</td>
</tr>
</tbody>
</table>

→ inherited
→ synthesized
S-attributed Grammars

- Special class of attribute grammars
- Only uses synthesized attributes (S-attributed)
- No use of inherited attributes

- Can be computed by any bottom-up parser during parsing
- Attributes can be stored on the parsing stack
- Reduce operation computes the (synthesized) attribute from attributes of children
### S-attributed Grammar: example

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → E ;</td>
<td>print(E.val)</td>
</tr>
<tr>
<td>E → E₁ + T</td>
<td>E.val = E₁.val + T.val</td>
</tr>
<tr>
<td>E → T</td>
<td>E.val = T.val</td>
</tr>
<tr>
<td>T → T₁ * F</td>
<td>T.val = T₁.val * F.val</td>
</tr>
<tr>
<td>T → F</td>
<td>T.val = F.val</td>
</tr>
<tr>
<td>F → (E)</td>
<td>F.val = E.val</td>
</tr>
<tr>
<td>F → digit</td>
<td>F.val = digit.lexval</td>
</tr>
</tbody>
</table>
example
L-attributed grammars

- L-attributed attribute grammar when every attribute in a production $A \rightarrow X_1...X_n$ is
  - A synthesized attribute, or
  - An inherited attribute of $X_j$, $1 \leq j \leq n$ that only depends on
    - Attributes of $X_1...X_{j-1}$ to the left of $X_j$, or
    - Inherited attributes of $A$
Summary

- Contextual analysis can move information between nodes in the AST
  - Even when they are not “local”
- Attribute grammars
  - Attach attributes and semantic actions to grammar
- Attribute evaluation
  - Build dependency graph, topological sort, evaluate
- Special classes with pre-determined evaluation order: S-attributed, L-attributed
The End
Identification
Scopes
Semantic Checks

- **Scope rules**
  - Use symbol table to check that
    - Identifiers defined before used
    - No multiple definition of same identifier
    - Program conforms to scope rules

- **Type checking**
  - Check that types in the program are consistent
  - How?
Type Checking

- Type rules specify
  - which types can be combined with certain operator
  - Assignment of expression to variable
  - Formal and actual parameters of a method call

- Examples

  ```
  string s1 = "drive";
  string s2 = "drink"
  int i = 42;
  string s3 = "the answer"

  s1 + s2; // No error
  s1 + i; // Type mismatch
  i + s3; // Type mismatch
  ```

  ERROR
Type Checking Rules

- Specify for each operator
  - Types of operands
  - Type of result

- Basic Types
  - Building blocks for the type system (type rules)
  - e.g., int, boolean, string

- Type Expressions
  - Array types
  - Function types
  - Record types / Classes
Typing Rules

If \( E_1 \) has type \( \text{int} \) and \( E_2 \) has type \( \text{int} \),
then \( E_1 + E_2 \) has type \( \text{int} \)

\[
\begin{align*}
E_1 : \text{int} & \quad E_2 : \text{int} \\
\hline
E_1 + E_2 : \text{int}
\end{align*}
\]

(Generally, also use a context \( A \))
## More Typing Rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Type</th>
<th>Type</th>
<th>Operation</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \vdash \text{true} : \text{boolean}$</td>
<td>$A \vdash \text{false} : \text{boolean}$</td>
<td>$A \vdash \text{int-literal} : \text{int}$</td>
<td>$A \vdash \text{string-literal} : \text{string}$</td>
<td></td>
</tr>
<tr>
<td>$A \vdash E_1 : \text{int}$</td>
<td>$A \vdash E_2 : \text{int}$</td>
<td>$\text{op} \in { +, -, /, *, %}$</td>
<td>$\text{op} \in { +, -, /, *, %}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A \vdash E_1 \text{ op E}_2 : \text{int}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A \vdash E_1 \text{ rop E}_2 : \text{boolean}$</td>
<td></td>
<td>$\text{rop} \in { &lt;=, &lt;, &gt;, &gt;=}$</td>
<td>$\text{rop} \in { ==, !=}$</td>
<td></td>
</tr>
<tr>
<td>$A \vdash E_1 : T$</td>
<td>$A \vdash E_2 : T$</td>
<td></td>
<td>$\text{rop} \in { ==, !=}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$A \vdash E_1 \text{ rop E}_2 : \text{boolean}$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
And Even More Typing Rules

\[
\begin{align*}
A \vdash E_1 : \text{boolean} & \quad A \vdash E_2 : \text{boolean} \\
\quad & \quad lop \in \{ \&\&, || \} \\
A \vdash E_1 \ lop \ E_2 : \text{boolean}
\end{align*}
\]

\[
\begin{align*}
A \vdash E_1 : \text{int} & \quad A \vdash E_1 : \text{boolean} \\
\quad & \quad A \vdash \neg E_1 : \text{int} \\
A \vdash ! E_1 : \text{boolean} \\
A \vdash E_1 : T[] & \quad A \vdash E_1 : T[] \quad A \vdash E_2 : \text{int} \\
\quad & \quad A \vdash E_1[E2] : T \\
A \vdash E_1.\text{length} : \text{int} & \quad A \vdash \text{new } T[E1] : T[]
\end{align*}
\]

\[
\begin{align*}
A \vdash T \in C & \quad \text{id} : T \in A \\
\quad & \quad A \vdash \text{new } T() : T \\
A \vdash \text{new } T() : T & \quad A \vdash \text{id} : T
\end{align*}
\]
Type Checking

- Our approach --- Traverse AST bottom-up and assign types for AST nodes
  - Use typing rules to compute node types
- More complicated alternative --- type-check during parsing
  - But naturally also more efficient
Example

45 > 32 && !false

A :- E1 : boolean  A :- E2 : boolean
A :- E1 lop E2 : boolean
lop ∈ { &&, || }
A :- E1 : boolean
A :- !E1 : boolean
A :- E1 : int  A :- E2 : int
A :- E1 rop E2 : boolean
rop ∈ { <=, <, >, >= }
A :- false : boolean
A :- int-literal : int