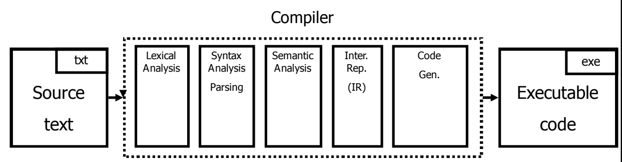
Lecture 05 – Syntax analysis & Semantic Analysis

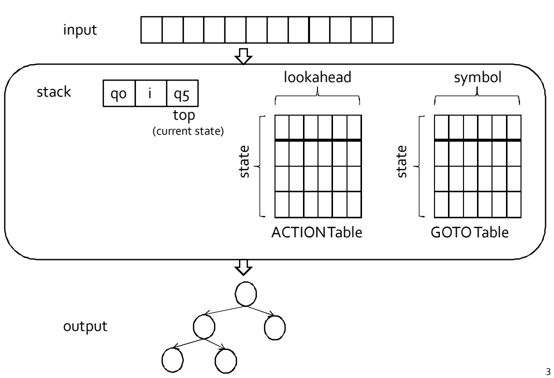
THEORY OF COMPILATION

Eran Yahav

You are here



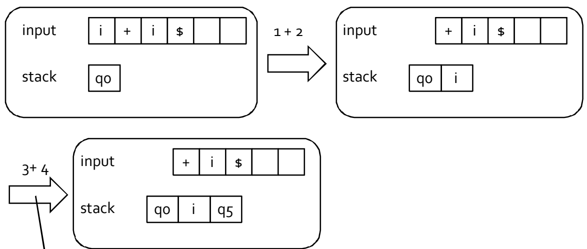
Last week: LR Parsing with Pushdown Automaton



Last week: LR Parsing with Pushdown Automaton

- s = top of stack, t = next token,
  use 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' = 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 α 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[q][N] table
  - Push new state s' on the stack (on top of N)

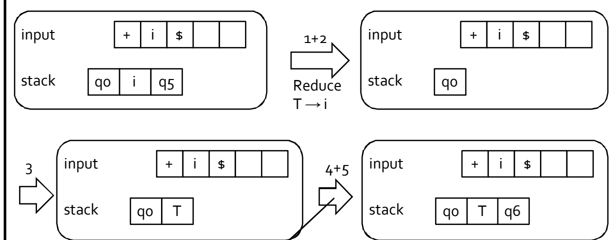
Last week: shift move



State	i	+	(	)	$	E	T	action
q0	q5		q7			q1	q6	shift

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



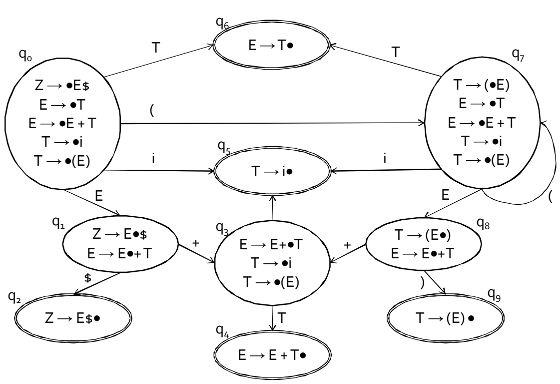
How we picked next state

State	i	+	(	)	$	E	T	action
q0	q5		q7			q1	q6	shift
…								
q5								r4

how we decided on a reduce

1. Using a rule N →α (ACTION[s][t])
2. Symbols in α and their following states are removed from stack. q = top afterwards.
3. Push N on the stack
4. New state s'= 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

State	i	+	(	)	$	E	T
q0	s5		s7			s1	s6
q1		s3			s2		
q2	r1	r1	r1	r1	r1	r1	r1
q3	s5		s7				s4
q4	r3	r3	r3	r3	r3	r3	r3
q5	r4	r4	r4	r4	r4	r4	r4
q6	r2	r2	r2	r2	r2	r2	r2
q7	s5		s7			s8	s6
q8		s3		s9			
q9	r5	r5	r5	r5	r5	r5	r5

```
(1) Z → E $
(2) E → T
(3) E → E + T
(4) T → i
(5) T →( 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 →α
  - Symbols in α 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

st	i	+	(	)	$	E	T
q0	s5		s7			s1	s6
q1		s3			s2		
q2	r1	r1	r1	r1	r1	r1	r1
q3	s5		s7				s4
q4	r3	r3	r3	r3	r3	r3	r3
q5	r4	r4	r4	r4	r4	r4	r4
q6	r2	r2	r2	r2	r2	r2	r2
q7	s5		s7			s8	s6
q8		s3		s9			
q9	r5	r5	r5	r5	r5	r5	r5

```
(1) Z → E $
(2) E → T
(3) E → E + T
(4) T → i
(5) T →( E )
```

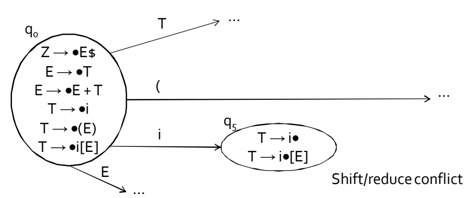
top is on the right

Stack	Input	Action
q0	i + i $	s5
q0 i q5	+ i $	r4
q0 T q6	+ i $	r2
q0 E q1	+ i $	s3
q0 E q1 + q3	i $	s5
q0 E q1 + q3 i q5	$	r4
q0 E q1 + q3 T q4	$	r3
q0 E q1	$	s2
q0 E q1 $ q2		r1
q0 Z		

Are we done?

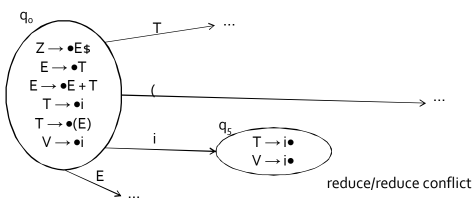
- Can make a transition diagram for any grammar
- Can make a GOTO table for every grammar
- Cannot make a deterministic ACTION table for every grammar

LR(0) Conflicts




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

LR(0) Conflicts




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


LR(0) Conflicts

- Any grammar with an ε-rule cannot be LR(0)
- Inherent shift/reduce conflict
  - A→ ε• - reduce item
  - P →α•Aβ – shift item
  - A→ ε• can always be predicted from P →α•Aβ

Back to the GOTO/ACTIONS tables

GOTO Table | ACTION Table

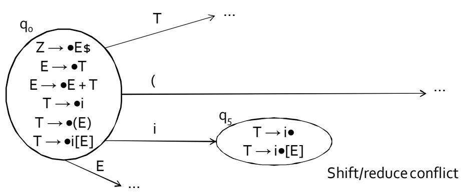
State	i	+	(	)	$	E	T	action
q0	q5		q7			q1	q6	shift
q1		q3			q2			shift
q2								Z→E$
q3	q5		q7				q4	Shift
q4								E→E+T
q5								T→i
q6								E→T
q7	q5		q7			q8	q6	shift
q8		q3		q9				shift
q9								T→E

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




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

input
```
A[x]$
```

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


SLR ACTION Table

State	i	+	(	)	$
q0	shift		shift		
q1		shift			shift
q2					Z→E$
q3	shift		shift		
q4		E→E+T		E→E+T	E→E+T
q5		T→i		T→i	T→i
q6		E→T		E→T	E→T
q7	shift		shift		
q8		shift		shift	
q9		T→(E)		T→(E)	T→(E)

Look-ahead token from the input

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

```
(1) Z → E $
(2) E → T
(3) E → E + T
(4) T → i
(5) T →( E )
```

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


SLR ACTION Table

missing rules

State	i	+	(	)	[	]	$
q0	shift		shift				
q1		shift					shift
q2							Z→E$
q3	shift		shift				
q4		E→E+T		E→E+T			E→E+T
q5		T→i		T→i	$\boxed{\text{shift}}$		T→i
q6		E→T		E→T			E→T
q7	shift		shift				
q8		shift		shift			
q9		T→(E)		T→(E)			T→(E)

SLR – use 1 token look-ahead

vs.

state	action
q0	shift
q1	shift
q2	Z→E$
q3	Shift
q4	E→E+T
q5	T→i
q6	E→T
q7	shift
q8	shift
q9	T→E

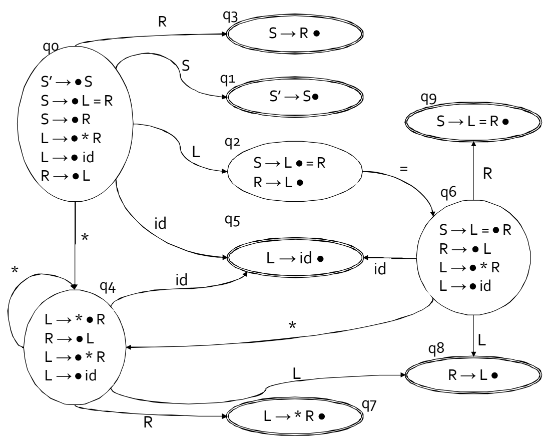
LR(0) – no look-ahead

```
… as before…
T → i
T → i[E]
```

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


Are we done?

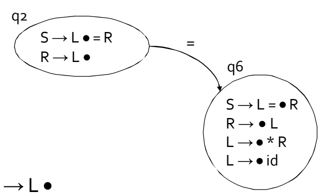
(0) S' → S
(1) S → L = R
(2) S → R
(3) L → * R
(4) L → id
(5) R → L



21

Shift/reduce conflict

(0) S' → S
(1) S → L = R
(2) S → R
(3) L → * R
(4) L → id
(5) R → L



- S → L • = R vs. R → L •
- FOLLOW(R) contains =
  - S ⇒ L = R ⇒ * R = R
- SLR cannot resolve the conflict either

22

LR(1) Grammars

- In SLR: a reduce item N → α• 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

23

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 → id yields the following LR(1) items

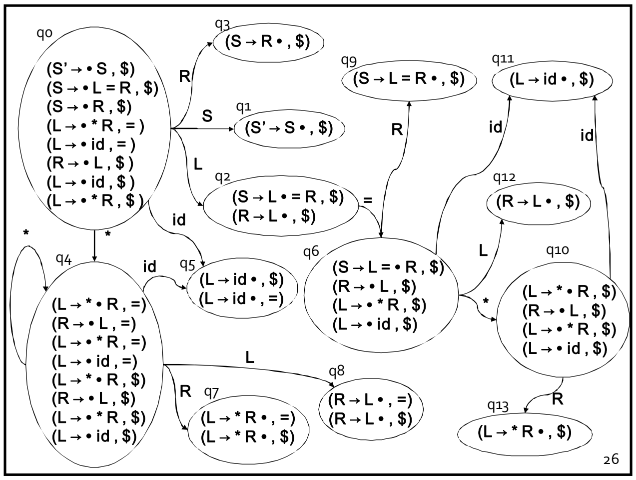
(0) S' → S
(1) S → L = R
(2) S → R
(3) L → * R
(4) L → id
(5) R → L

[L → • id, *]
[L → • id, =]
[L → • id, id]
[L → • id, $]
[L → id •, *]
[L → id •, =]
[L → id •, id]
[L → id •, $]

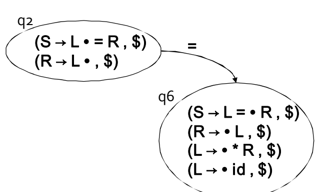
24

ε-closure for LR(1)

- For every [A → α ● Bβ , c] in S
  - for every production B→δ and every token b in the grammar such that b ∈ FIRST(βc)
  - Add [B → ● δ , 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 σ (the look-ahead set computed for the item)

29

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

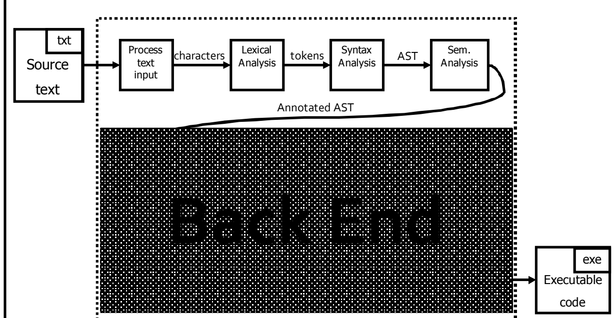
30

Summary

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

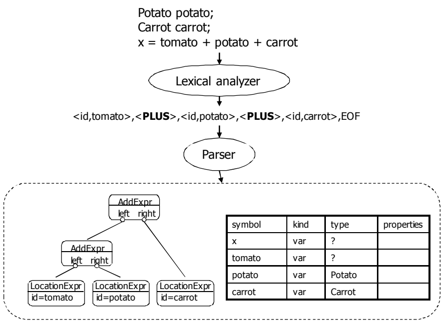
31

You are here…



32

What we want



symbol	kind	type	properties
x	var	?	
tomato	var	?	
potato	var	Potato	
carrot	var	Carrot	

tomato is undefined
potato used before initialized
Cannot add Potato and Carrot

33

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"

34

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

35

Abstract Syntax Tree (AST)

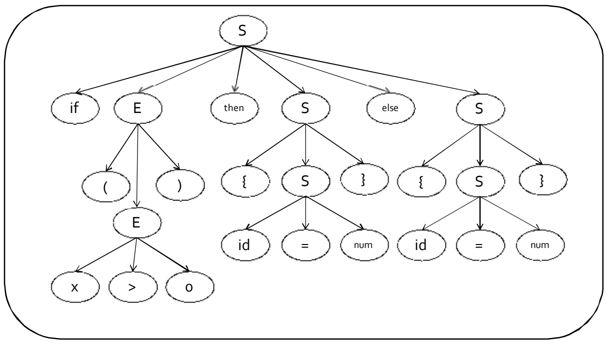
- Abstract away some syntactic details of the source language

```
S → if E then S else S
  | …
```

```
if (x>0)
then { y = 42}
else { y = 73 }
```

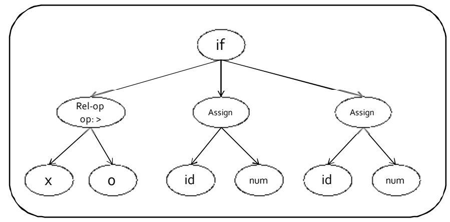
36

Parse tree (concrete syntax tree)



37

Abstract Syntax Tree (AST)



38

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

39

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:
      Expr ⟶ Expr + Term
      Expr.type = Expr1.type when (Expr1.type == Term.type)
      Error otherwise

40

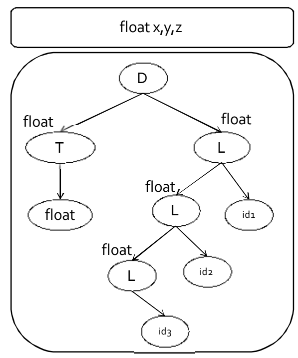
Indexed symbols

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

- Expr ⟶ Expr + Term
  Becomes
  Expr ⟶ Expr1 + Term

41

Example



Production	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)

42

Dependencies

- A semantic equation a = b1,…,bm requires computation of b1,…,bm to determine the value of a

- The value of a depends on b1,…,bm
  - We write a ⟵ bi

43

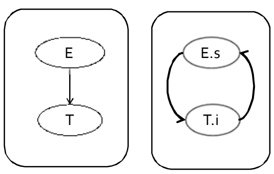
Attribute Evaluation

- 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

44

Cycles

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



AST

Dependence graph

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

45

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

46

Building Dependency Graph

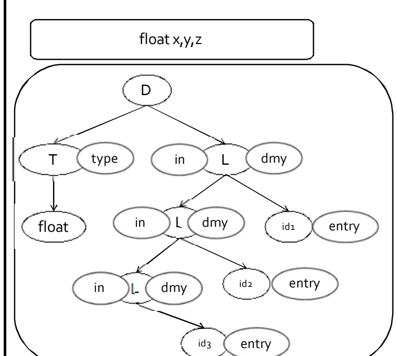
- All semantic equations take the form

  attr1 = func1(attr1.1, attr1.2,…)
  attr2 = func2(attr2.1, attr2.2,…)

- 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(c1,c2,…ck) add edges of the form (ci,b)

47

Example

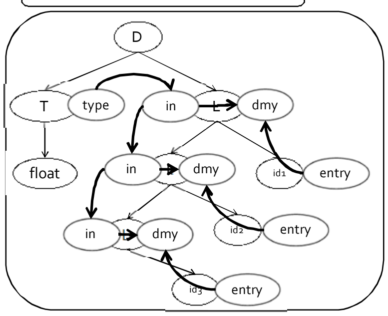


Prod.	Semantic Rule
D → T L	L.in = T.type
T → int	T.type = integer
T → float	T.type = float
L → L1, id	L1.in = L.in addType(id.entry,L.in)
L → id	addType(id.entry,L.in)

48

Example

float x,y,z

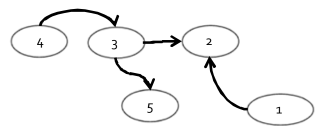


Prod.	Semantic Rule
D → T L	L.in = T.type
T → int	T.type = integer
T → float	T.type = float
L → L1, id	L1.in = L.in addType(id.entry,L.in)
L → id	addType(id.entry,L.in)

49

Topological Order

- For a graph G=(V,E), |V|=k
- Ordering of the nodes v1,v2,…vk such that for every edge (vi,vj) ∈ E, i < j

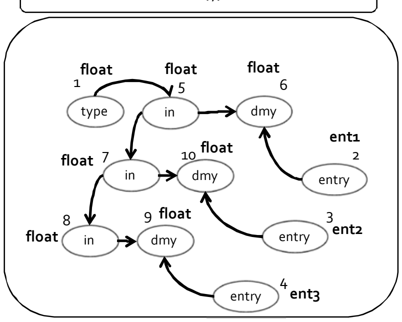


Example topological orderings: 1 4 3 2 5, 4 1 3 5 2

50

Example

float x,y,z



51

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

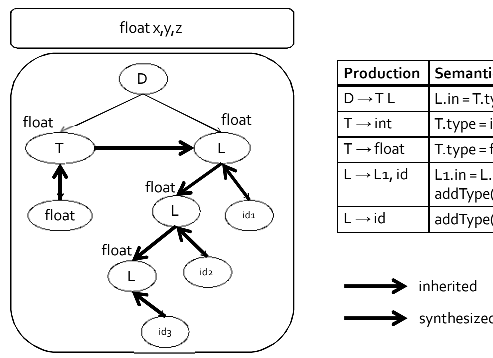
52

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

53

example



Production	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)

54

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

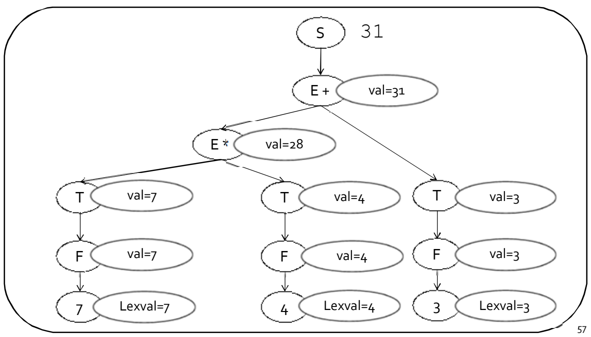
55

S-attributed Grammar: example

Production	Semantic Rule
S→ E ;	print(E.val)
E → E1 + T	E.val = E1.val + T.val
E → T	E.val = T.val
T → T1 * F	T.val = T1.val * F.val
T → F	T.val = F.val
F → (E)	F.val = E.val
F → digit	F.val = digit.lexval

56

example



57

L-attributed grammars

- L-attributed attribute grammar when every attribute in a production $A \rightarrow X1 \ldots Xn$ is
  - A synthesized attribute, or
  - An inherited attribute of Xj, 1 <= j <=n that only depends on
    - Attributes of X1…Xj-1 to the left of Xj, or
    - Inherited attributes of A

58

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

59

The End

60

Identification

61

Scopes

62

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?

63

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 string
"drive" + "drink"

string

int string
42 + "the answer"

ERROR

64

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

65

Typing Rules

If E1 has type int and E2 has type int,
then E1 + E2 has type int

$$\frac{E1 : int \qquad E2 : int}{E1 + E2 : int}$$

(Generally, also use a context A)

66

More Typing Rules

$$\frac{}{A \vdash true : boolean} \qquad \frac{}{A \vdash false : boolean}$$

$$\frac{}{A \vdash \textit{int-literal} : int} \qquad \frac{}{A \vdash \textit{string-literal} : string}$$

$$\frac{A \vdash E1 : int \qquad A \vdash E2 : int}{A \vdash E1 \; \textit{op} \; E2 : int} \qquad \textit{op} \in \{ +, -, /, *, \% \}$$

$$\frac{A \vdash E1 : int \qquad A \vdash E2 : int}{A \vdash E1 \; \textit{rop} \; E2 : boolean} \qquad \textit{rop} \in \{ <=, <, >, >= \}$$

$$\frac{A \vdash E1 : T \qquad A \vdash E2 : T}{A \vdash E1 \; \textit{rop} \; E2 : boolean} \qquad \textit{rop} \in \{ ==, != \}$$

67

And Even More Typing Rules

$$\frac{A \vdash E1 : boolean \qquad A \vdash E2 : boolean}{A \vdash E1 \; \textit{lop} \; E2 : boolean} \qquad \textit{lop} \in \{ \&\&, || \}$$

$$\frac{A \vdash E1 : int}{A \vdash \mathbf{-} \, E1 : int} \qquad \frac{A \vdash E1 : boolean}{A \vdash \mathbf{!} \, E1 : boolean}$$

$$\frac{A \vdash E1 : T[]}{A \vdash E1.length : int} \qquad \frac{A \vdash E1 : T[] \qquad A \vdash E2 : int}{A \vdash E1[E2] : T} \qquad \frac{A \vdash E1 : int}{A \vdash new \; T[E1] : T[]}$$

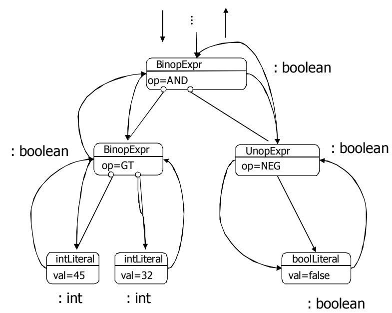
$$\frac{A \vdash T \text{ \\in } C}{A \vdash new \; T() : T} \qquad \frac{id : T \in A}{A \vdash id : T}$$

68

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

$$\frac{A :- E1 : boolean \quad A :- E2 : boolean}{A :- E1 \; lop \; E2 : boolean} \quad lop \in \{ \&\&, || \}$$

$$\frac{A :- E1 : boolean}{A :- !E1 : boolean}$$

$$\frac{A :- E1 : int \quad A :- E2 : int}{A :- E1 \; rop \; E2 : boolean} \quad rop \in \{ <=, <, >, >= \}$$

$$\frac{}{A :- false : boolean}$$

$$\frac{}{A :- \textit{int-literal} : int}$$