

Attribute Grammars

- Pagan Ch. 2.1, 2.2, 2.3, 3.2
- Stansifer Ch. 2.2, 2.3
- Slonneger and Kurtz Ch 3.1, 3.2

1

Formal Languages

- Important role in the design and implementation of programming languages
- **Alphabet**: finite set Σ of symbols
- **String**: finite sequence of symbols
 - Empty string ϵ
 - Σ^* - set of all strings over Σ (incl. ϵ)
 - Σ^+ - set of all non-empty strings over Σ
- **Language**: set of strings $L \subseteq \Sigma^*$

2

Grammars

- $G = (N, T, S, P)$
 - Finite set of **non-terminal symbols** N
 - Finite set of **terminal symbols** T
 - Starting non-terminal symbol $S \in N$
 - Finite set of **productions** P
- Production: $x \rightarrow y$
 - $x \in (N \cup T)^+$, $y \in (N \cup T)^*$
- Applying a production: $uxv \Rightarrow uyw$

3

Languages and Grammars

- String derivation
 - $w_1 \Rightarrow w_2 \Rightarrow \dots \Rightarrow w_n$; denoted $w_1 \xRightarrow{*} w_n$
- Language generated by a grammar
 - $L(G) = \{ w \in T^* \mid S \xRightarrow{*} w \}$
- Traditional classification
 - Regular
 - Context-free
 - Context-sensitive
 - Unrestricted

4

Regular Languages

- Generated by **regular grammars**
 - All productions are $A \rightarrow wB$ and $A \rightarrow w$
 - $A, B \in N$ and $w \in T^*$
 - Or all productions are $A \rightarrow Bw$ and $A \rightarrow w$
- e.g. $L = \{ a^n b \mid n > 0 \}$ is a regular language
 - $S \rightarrow Ab$ and $A \rightarrow a \mid Aa$
- Alternative equivalent formalisms
 - Regular expressions: e.g. a^*b for $\{ a^n b \mid n \geq 0 \}$
 - Deterministic finite automata (DFA)
 - Nondeterministic finite automata (NFA)

5

Uses of Regular Languages

- **Lexical analysis** in compilers
 - e.g. identifier = letter (letter|digit)*
 - Sequence of tokens for the **syntactic analysis** done by the parser
 - tokens = terminals for the context-free grammar of the parser
- Pattern matching
 - `grep "a\+b" foo.txt`
 - Every line from foo.txt that contains a string from the language $L = \{ a^n b \mid n > 0 \}$
 - i.e. the language for reg. expr. a^+b

6

Context-Free Languages

- Subsume regular languages
 - $L = \{ a^n b^n \mid n > 0 \}$ is c.f. but not regular
- Generated by a **context-free grammar**
 - Each production: $A \rightarrow w$
 - $A \in N, w \in (N \cup T)^*$
- BNF: alternative notation for context-free grammars
 - Backus-Naur form: John Backus and Peter Naur, for ALGOL60

7

BNF Example

```
<stmt> ::= while <exp> do <stmt>
         | if <exp> then <stmt>
         | if <exp> then <stmt> else <stmt>
         | <exp> := <exp>
         | <id> ( <exps> )
<exps> ::= <exp> | <exps> , <exp>
```

8

EBNF Example

```
<stmt> ::= while <exp> do <stmt>
         | if <exp> then <stmt> [ else <stmt> ]
         | <exp> := <exp>
         | <id> ( <exp> { , <exp> } )
```

Extensions

- [...] : optional sequence of symbols
- { ... } : repeated zero or more times

9

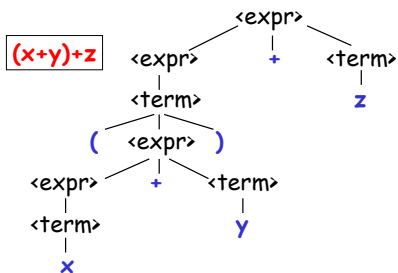
Derivation Tree

- Also called **parse tree** or **concrete syntax tree**
 - Leaf nodes: terminals
 - Inner nodes: non-terminals
 - Root: starting non-terminal of the grammar
- Describes a particular way to derive a string
 - Leaf nodes from left to right are the string
 - to get the string: depth-first traversal, following the leftmost unexplored branch

10

Example of a Derivation Tree

$\langle \text{expr} \rangle ::= \langle \text{term} \rangle \mid \langle \text{expr} \rangle + \langle \text{term} \rangle$
 $\langle \text{term} \rangle ::= x \mid y \mid z \mid (\langle \text{expr} \rangle)$



11

Derivation Sequences

- Each tree represents a **set** of derivation sequences
 - Differ in the order of production application
- The tree "filters out" the choice of order of production application
- Filtering out the order
 - Parse tree
 - Leftmost derivation: always replace the leftmost non-terminal
 - Rightmost derivation: ... rightmost ...

12

Equivalent Derivation Sequences

The set of string derivations that are represented by the same parse tree

One derivation:

$\langle \text{expr} \rangle \Rightarrow \langle \text{expr} \rangle + \langle \text{term} \rangle \Rightarrow \langle \text{expr} \rangle + z \Rightarrow$
 $\langle \text{term} \rangle + z \Rightarrow (\langle \text{expr} \rangle) + z \Rightarrow$
 $(\langle \text{expr} \rangle + \langle \text{term} \rangle) + z \Rightarrow (\langle \text{expr} \rangle + y) + z \Rightarrow$
 $(\langle \text{term} \rangle + y) + z \Rightarrow (x + y) + z$

Another derivation:

$\langle \text{expr} \rangle \Rightarrow \langle \text{expr} \rangle + \langle \text{term} \rangle \Rightarrow \langle \text{term} \rangle + \langle \text{term} \rangle \Rightarrow$
 $(\langle \text{expr} \rangle) + \langle \text{term} \rangle \Rightarrow (\langle \text{expr} \rangle + \langle \text{term} \rangle) + \langle \text{term} \rangle \Rightarrow$
 $(\langle \text{term} \rangle + \langle \text{term} \rangle) + \langle \text{term} \rangle \Rightarrow (x + \langle \text{term} \rangle) + \langle \text{term} \rangle \Rightarrow$
 $(x + y) + \langle \text{term} \rangle \Rightarrow (x + y) + z$

Many more ...

13

Ambiguous Grammars

- For some string, there are two different parse trees
 - i.e. two different leftmost derivations
 - i.e. two different rightmost derivations
- For programming languages, we typically have non-ambiguous grammars
 - Need to build parsers
 - Add non-terminals to remove ambiguity
 - Operator precedence and associativity

14

Use of Context-Free Grammars

- **Syntax** of a programming language
 - e.g. Java: Chapter 18 of the language specification (JLS) defines a grammar
 - Terminals: identifiers, keywords, literals, separators, operators
 - Starting non-terminal: **CompilationUnit**
- Implementation of a **parser** in a compiler
 - Syntactic analysis: takes a compilation unit and produces a parse tree
 - e.g. the JLS grammar (Ch. 18) is used by the parser in Sun's **javac** compiler

15

Limitations of Context-Free Grammars

- Cannot represent semantics
- e.g. "every variable used in a statement should be declared in advance"
- e.g. "the use of a variable should conform to its type" (type checking)
 - cannot say "string s1 divided by string s2"
- Solution: attribute grammars
 - For certain kinds of **semantic analysis**

16

Attribute Grammars

- Context-free grammar (BNF)
- Finite set of **attributes**
 - For each attribute: domain of possible values
 - For each terminal and non-terminal: set of associated attributes (may be empty)
 - Inherited or synthesized
- Set of **evaluation rules**
- Set of boolean **conditions** for attribute values

17

Example

- $L = \{ a^n b^n c^n \mid n > 0 \}$; not context-free
- BNF
 - $\langle \text{start} \rangle ::= \langle A \rangle \langle B \rangle \langle C \rangle$ $\langle A \rangle ::= a \mid a \langle A \rangle$
 - $\langle B \rangle ::= b \mid b \langle B \rangle$ $\langle C \rangle ::= c \mid c \langle C \rangle$
- Attributes
 - **Na**: associated with $\langle A \rangle$
 - **Nb**: associated with $\langle B \rangle$
 - **Nc**: associated with $\langle C \rangle$
 - Value domain = integers

18

Example

- Evaluation rules (similar for $\langle B \rangle$, $\langle C \rangle$)

$\langle A \rangle ::= a$

$Na(\langle A \rangle) := 1$

| $a\langle A \rangle_2$

$Na(\langle A \rangle) := 1 + Na(\langle A \rangle_2)$

- Conditions

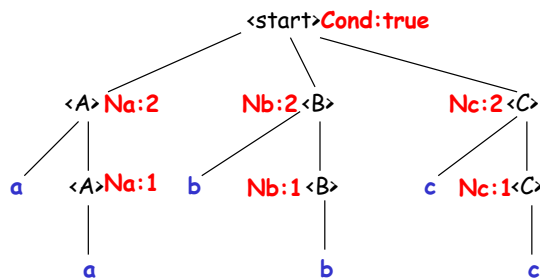
$\langle \text{start} \rangle ::= \langle A \rangle \langle B \rangle \langle C \rangle$

$\text{Cond}: Na(\langle A \rangle) = Nb(\langle B \rangle) = Nc(\langle C \rangle)$

- Alternative notation: $\langle A \rangle.Na$

19

Parse Tree



20

Parse Tree for an Attribute Grammar

- Valid tree for the underlying BNF
- Each node has a set of (attribute,value) pairs
 - One pair for each attribute associated with the terminal or non-terminal in the node
- Some nodes have boolean conditions
- **Valid** parse tree
 - Attribute values conform to the evaluation rules
 - All boolean conditions are true

21

Example: Ada Block Statement

x: begin a := 1; b := 2; end x;

- $\langle \text{block} \rangle ::= \langle \text{block id} \rangle_1 : \text{begin}$
 $\langle \text{stmts} \rangle \text{end} \langle \text{block id} \rangle_2 ;$
- Cond: $\text{value}(\langle \text{block id} \rangle_1) = \text{value}(\langle \text{block id} \rangle_2)$
- $\langle \text{stmts} \rangle ::= \langle \text{stmt} \rangle \mid \langle \text{stmts} \rangle \langle \text{stmt} \rangle$
- $\langle \text{block id} \rangle ::= \text{id}$
 - $\text{value}(\langle \text{block id} \rangle) := \text{name}(\text{id})$

22

Alternative

- Use a boolean attribute instead of the condition
- $\langle \text{block} \rangle.\text{OK} :=$
 $\langle \text{block id} \rangle_1.\text{value} = \langle \text{block id} \rangle_2.\text{value}$
- A valid parse tree must have $\langle \text{block} \rangle.\text{OK} = \text{true}$ for all block nodes

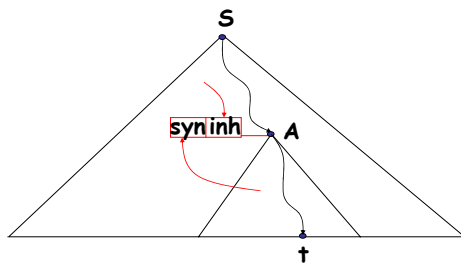
23

Synthesized vs. Inherited Attributes

- Synthesized attributes: computed using values from tree descendants
 - Production: $\langle A \rangle ::= \dots$
 - Evaluation rule: $\langle A \rangle.\text{syn} := \dots$
- Inherited: values from the parent node
 - Production: $\langle B \rangle ::= \dots \langle A \rangle \dots$
 - Evaluation rule: $\langle A \rangle.\text{inh} := \dots$
- In both cases, the evaluation rules can be arbitrarily complex: e.g. we could even use external "helper" functions

24

Synthesized vs. Inherited



25

Evaluation Rules

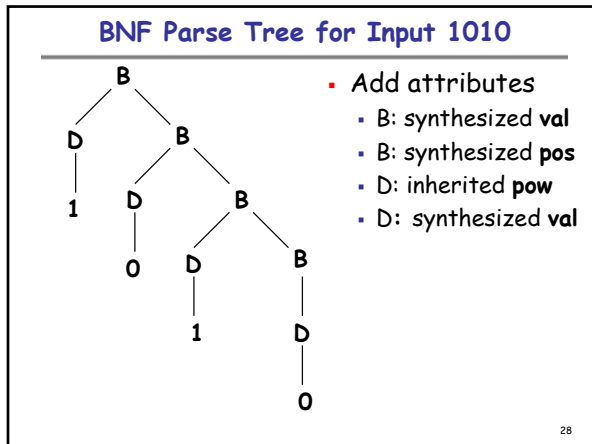
- Synthesized attribute associated with N:
 - Each alternative in N's production should contain a rule for evaluating the attribute
- Inherited attribute associated with N:
 - for every occurrence of N on the right-hand side of any alternative, there must be a rule for evaluating the attribute

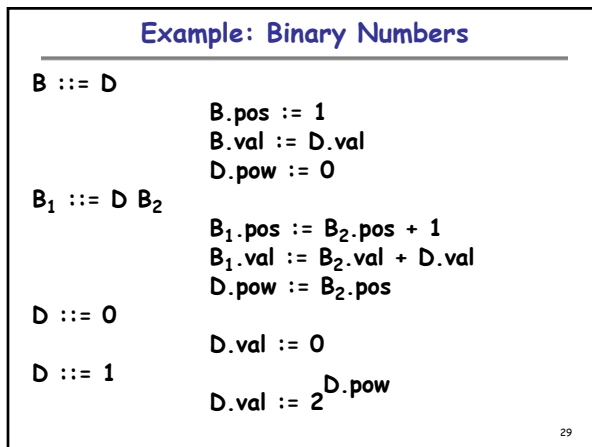
26

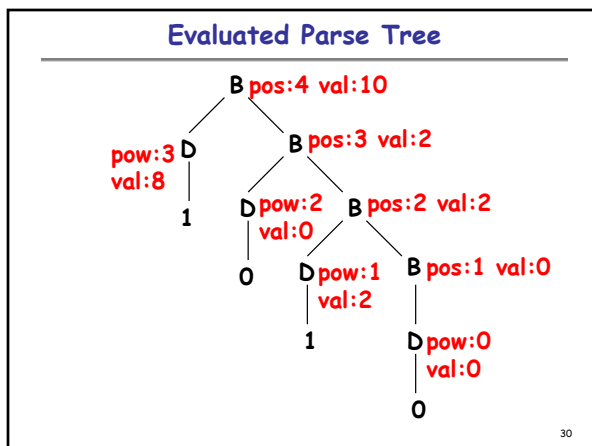
Example: Binary Numbers

- Context-free grammar
 - For simplicity, will use X instead of $\langle X \rangle$
 - $B ::= D$
 - $B ::= D B$
 - $D ::= 0$
 - $D ::= 1$
- Goal: compute the **value** of a binary number

27







Example: Expression Language

- Problem: evaluate an expression
- Syntax:
S ::= E
E ::= 0 | 1 | I | (E + E) | let I = E in E end
I ::= id
- Attributes
 - I: synthesized **name**
 - E: synthesized **val**, inherited **env**

31

Attribute Grammar

```
S ::= E
    E.env := EmptyEnvironment()
E ::= 0
    E.val := 0
E ::= 1
    E.val := 1
E ::= I
    E.val := lookup(E.env, I.name)
I ::= id
    I.name := id
```

32

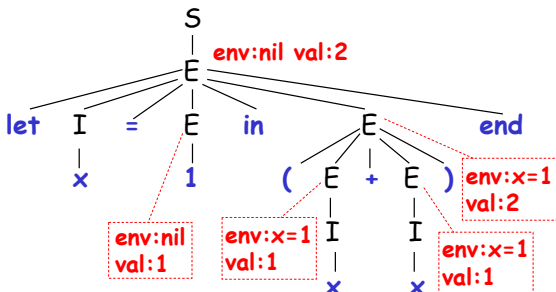
Attribute Grammar

```
E1 ::= (E2 + E3)
    E1.val := E2.val + E3.val
    E2.env := E1.env
    E3.env := E1.env
E1 ::= let I = E2 in E3 end
    E2.env := E1.env
    E3.env := update(E1.env, I.name, E2.val)
    E1.val := E3.val
```

33

Example

- Evaluation of `let x = 1 in (x+x) end`



34

More than Context-Free Power

- $L = \{ a^n b^n c^n \mid n > 0 \}$
 - Unlike $L = \{ a^n b^n \mid n > 0 \}$, here we need explicit counting
- $L = \{ w c w \mid w \in \{a,b\}^* \}$
 - The "flavor" of checking whether identifiers are declared before their uses
 - Cannot be done with a context-free grammar
- Syntax analysis (i.e. parser) cannot handle semantic properties

35

"Fixing" Context-Free Grammars

- $\langle S \rangle_1 ::= a \mid b \mid \langle S \rangle_2 \langle S \rangle_3$
 - Ambiguous: e.g. `abab` can be parsed as `(ab)(ab)` or `a(b(a(b)))` or ...
 - Suppose we want to make the grammar unambiguous, and associate to the right
- One approach: $X ::= a \mid b$ and $S ::= XS$
- Another approach: attribute `len` for S
 - $\langle S \rangle ::= a \mid b \quad \langle S \rangle.\text{len} := 1$
 - $\langle S \rangle_1 ::= \langle S \rangle_2 \langle S \rangle_3 \quad \langle S \rangle_1.\text{len} := \langle S \rangle_2.\text{len} + \langle S \rangle_3.\text{len}$
Cond: $\langle S \rangle_2.\text{len} = 1$

36

"Fixing" Context-Free Grammars

- $\langle E \rangle_1 ::= \langle \text{Num} \rangle \mid \langle E \rangle_2 + \langle E \rangle_3$ - ambiguous
- Want left-associativity:
 - Parse $a+b+c$ as $(a+b)+c$ instead of $a+(b+c)$
- Change BNF: $\langle E \rangle_1 ::= \langle \text{Num} \rangle \mid \langle E \rangle_2 + \langle \text{Num} \rangle$
- Alternative: attribute grammar
 - Attribute n : number of $+$
 - $\langle E \rangle_1 ::= \langle \text{Num} \rangle \quad \langle E \rangle_{1,n} := 0$
 - $\langle E \rangle_1 ::= \langle E \rangle_2 + \langle E \rangle_3 \quad \langle E \rangle_{1,n} := 1 + \langle E \rangle_{2,n} + \langle E \rangle_{3,n}$
 - Cond: ?

37

Attribute Grammar Evaluation

- Problem: arbitrary dependencies
 - $\langle A \rangle ::= \dots \langle B \rangle \dots$
 - $\langle A \rangle.s := \langle A \rangle.i$
 - $\langle B \rangle.i := \langle B \rangle.s$
- Solution: sort attributes by their dependencies, and use this as the evaluation order

38

Dependencies

- $\langle A \rangle.x := \langle B \rangle.y + \langle C \rangle.z$
 - $\langle A \rangle.x$ depends on $\langle B \rangle.y$
 - $A.x \leftarrow B.y$
 - $\langle A \rangle.x$ depends on $\langle C \rangle.z$
 - $A.x \leftarrow C.z$
- $\langle A \rangle_1.x := \langle A \rangle_2.x$
 - $\langle A \rangle_1.x$ depends on $\langle A \rangle_2.x$
 - $\langle A \rangle_1.x \leftarrow \langle A \rangle_2.x$

39

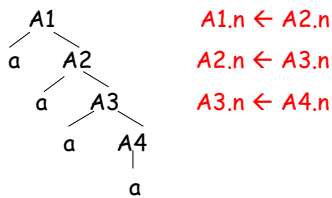
Dependencies

- It gets complicated with recursion:
 $\langle A \rangle_1 ::= a$
 $\langle A \rangle_{1.n} = 1$
 $| a \langle A \rangle_2$
 $\langle A \rangle_{1.n} = \langle A \rangle_{2.n} + 1$
- Dependency $A1.n \leftarrow A2.n$ isn't enough

40

Dependencies

- Example: input `aaaa`



- Need a global numbering of tree nodes

41

Attribute Grammar Evaluation

- Given: Parse tree for parsed input with attributes attached to tree nodes
- Find evaluation order of attributes
 - Globally "number" tree nodes
 - Build dependency graph
 - Complain about cycles in graph
 - Topologically sort the graph
- Evaluate the attributes in sorted order

42

Example: Binary Numbers

```

B ::= D
    B.pos := 1
    B.val := D.val
    D.pow := 0

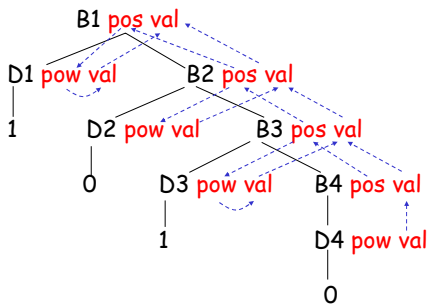
B1 ::= D B2
    B1.pos := B2.pos + 1
    B1.val := B2.val + D.val
    D.pow := B2.pos

D ::= 0
    D.val := 0

D ::= 1
    D.val := 2D.pow
    
```

43

Dependency Graph for Binary Numbers



44

Sort the Graph

- Topological sort: x is "smaller" than y iff $x \rightarrow y$

```

D4.pow, B4.pos, D4.val, B4.val,
B3.pos, D3.pow, D3.val, B3.val,
B2.pos, D2.pow, D2.val, B2.val,
B1.pos, D1.pow, D1.val, B1.val
    
```

45

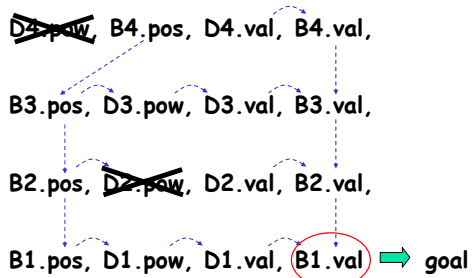
Cycles

- The notion of “topological sort” only makes sense for directed acyclic graphs
- If we have cycles in the dependence graph: recursive dependence between a set of attributes
 - There are approaches to solve recursive systems of equations
 - e.g. fixed-point computation
- For the rest of the discussion, we will disallow cycles

46

Optimizations

- Remove nodes based on reachability



47

Example: Type Checking

- Given: program with declarations
- Check that variables are used correctly

```
begin  
  bool i;  
  int j;  
  begin  
    int i;  
    x := i + j;  
  end  
end
```

48

Context-Free Grammar

```
<prog> ::= <block>
<block> ::= begin <decls> ; <stmts> end
<stmts> ::= <stmt>
           | <stmt> ; <stmts>
<stmt> ::= <assign>
          | <block>
          | ...
...
```

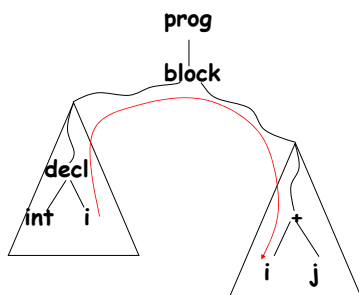
49

Problem

- Check types of variables (int, bool)
- For nested blocks use innermost declaration (static scoping)
- Check parameter types and return type for functions
 - All functions have return type int

50

Parse Tree



51

Data Structure

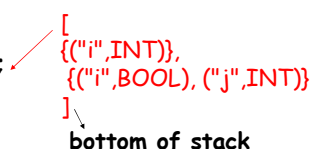
- Use a **stack** of **set of name-type pairs**
 - symbol table: set of name-type pairs
- Build a symbol table for the declarations in a block
 - as a synthesized attribute **tbl**
- Use stack of symbol tables in statements and expressions
 - as an inherited attribute **symtab**

52

Example: Type Checking

- Pass around stack of symbol tables

```
begin
  bool i;
  int j;
  begin
    int i;
    x := i + j;
  end
end
```



53

Attribute Grammar

```
<prog> ::= <block>
         <block>.symtab := emptystack
<block> ::= begin <decls> ; <stmts> end
         <stmts>.symtab :=
           push(<decls>.tbl, <block>.symtab)
<stmts>1 ::= <stmt>
         <stmt>.symtab := <stmts>1.symtab
         | <stmt> ; <stmts>2
         <stmt>.symtab := <stmts>1.symtab
         <stmts>2.symtab := <stmts>1.symtab
```

54

Declarations

```
<decls>1 ::= <decl>
  <decls>1.tbl := <decl>.tbl
  | <decl> ; <decls>2
  <decls>1.tbl :=
    <decl>.tbl ∪ <decls>2.tbl
  Cond:
    ids(<decl>.tbl) ∩ ids(<decls>2.tbl) = ∅
```

ids is a function that takes a set of name-type pairs and returns the set of all names

55

Declarations

```
<decl> ::= int <id>
  <decl>.tbl := { (<id>.name, INT) }
  | bool <id>
  <decl>.tbl := { (<id>.name, BOOL) }
  | fun <id> ( <params> ) : int = <block>
  <decl>.tbl :=
    { (<id>.name,
      synthesized attr:
      ordered list of INT/BOOL
      FUN(<params>.types, INT) ) }
  <block>.symtab := Oops!
```

56

Back to the Drawing Board

- Declarations: add inherited attr **symtab**

```
<block> ::= begin <decls> ; <stmts> end
  <stmts>.symtab :=
    push(<decls>.tbl, <block>.symtab)
  <decls>.symtab :=
    push(<decls>.tbl, <block>.symtab)
```

We first gather the declarations in <decls>, and then we use them to check the blocks "embedded" in <decls>

57

Declarations

```
<decls>1 ::= <decl>
  <decls>1.tbl := <decl>.tbl
  | <decl> ; <decls>2
  <decls>1.tbl :=
    <decl>.tbl ∪ <decls>2.tbl
  <decl>.symtab := <decls>1.symtab
  <decls>2.symtab := <decls>1.symtab
Cond:
  ids(<decl>.tbl) ∩ ids(<decls>2.tbl) = ∅
```

58

Declarations

```
<decl> ::= int <id>
  <decl>.tbl := { (<id>.name, INT) }
  | bool <id>
  <decl>.tbl := { (<id>.name, BOOL) }
  | fun <id> ( <params> ) : int = <block>
  <decl>.tbl :=
    { (<id>.name,
      FUN(<params>.types, INT) ) }
  <block>.symtab :=
    push(<params>.tbl, <decl>.symtab)
```

59

Type-Checking Function Bodies

- Is the following code legal?

```
fun f (int i): int =
  begin
    ... g(5) ...
  end;
fun g (int j): int =
  begin
    ...
  end
```

60

Statements

```
<stmts>1 ::= <stmt>
  <stmt>.syntab := <stmts>1.syntab
  | <stmt> ; <stmts>
  <stmt>.syntab := <stmts>1.syntab
  <stmts>2.syntab := <stmts>1.syntab
<stmt> ::= <assign>
  <assign>.syntab := <stmt>.syntab
  | <block>
  <block>.syntab := <stmt>.syntab
  | if <boolexp> then <stmts> else ...
  <boolexp>.syntab := <stmt>.syntab
  <stmts>.syntab := <stmt>.syntab
```

61

Statements

```
<assign> ::= <id> := <intexp>
  <intexp>.syntab := <assign>.syntab
  Cond:
  the "last" type on the stack ← typeof(<id>.name, <assign>.syntab) = INT
  | <id> := <boolexp>
  <boolexp>.syntab := <assign>.syntab
  Cond:
  typeof(<id>.name, <assign>.syntab) = BOOL
```

62

Expressions

```
<intexp>1 ::= <integer>
  | <id>
  Cond: typeof(<id>.name, <intexp>1.syntab) = INT
  | <intexp>2 + <intexp>3
  <intexp>2.syntab := <intexp>1.syntab
  <intexp>3.syntab := <intexp>1.syntab
<boolexp> ::= true
  | false
  | <id>
  Cond: typeof(<id>.name, <boolexp>.syntab) = BOOL
```

63

Function Call

$\langle \text{intexp} \rangle ::= \langle \text{id} \rangle (\langle \text{args} \rangle)$
 Cond: $\text{typeof}(\langle \text{id} \rangle.\text{name}, \langle \text{intexp} \rangle.\text{symtab}) = \text{FUN}$
 redundant if we have only int ret types
 Cond: $\text{rettype}(\langle \text{id} \rangle.\text{name}, \langle \text{intexp} \rangle.\text{symtab}) = \text{INT}$
 list of $\langle \text{intexp} \rangle$ and $\langle \text{boolexp} \rangle$
 $\langle \text{args} \rangle.\text{expTypes} := \text{paramtypes}(\langle \text{id} \rangle.\text{name}, \langle \text{intexp} \rangle.\text{symtab})$
 inherited attr: list of BOOL/INT
 $\langle \text{args} \rangle.\text{symtab} := \langle \text{intexp} \rangle.\text{symtab}$

64

Example: Code Generation

- Given: parse tree for a simple program (after type checking)
- Translate to assembly code
- The evaluation rules of the attribute grammar generate the assembly code

65

Simple Imperative Language (IMP)

$\langle c \rangle_1 ::= \text{skip} \mid \langle \text{id} \rangle := \langle \text{ae} \rangle \mid \langle c \rangle_2 ; \langle c \rangle_3$
 $\mid \text{if } \langle \text{be} \rangle \text{ then } \langle c \rangle_2 \text{ else } \langle c \rangle_3$
 $\mid \text{while } \langle \text{be} \rangle \text{ do } \langle c \rangle_2$
 $\langle \text{ae} \rangle_1 ::= \langle \text{id} \rangle \mid \langle \text{int} \rangle \mid \langle \text{ae} \rangle_2 + \langle \text{ae} \rangle_3$
 $\mid \langle \text{ae} \rangle_2 - \langle \text{ae} \rangle_3 \mid \langle \text{ae} \rangle_2 * \langle \text{ae} \rangle_3$
 $\langle \text{be} \rangle_1 ::= \text{true} \mid \text{false}$
 $\mid \langle \text{ae} \rangle_1 = \langle \text{ae} \rangle_2 \mid \langle \text{ae} \rangle_1 < \langle \text{ae} \rangle_2$
 $\mid \neg \langle \text{be} \rangle_2 \mid \langle \text{be} \rangle_2 \wedge \langle \text{be} \rangle_3$
 $\mid \langle \text{be} \rangle_2 \vee \langle \text{be} \rangle_3$

66

Assembly Language

- Processor with a single register
 - Accumulator
- **LOAD x**: copy the value of memory location (variable) x into the accumulator
- **LOAD const**: set the value of the accumulator to an integer constant
- **STO x**: write accumulator to memory location (variable) x

67

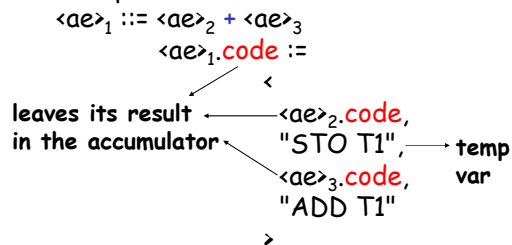
Assembly Language

- **ADD x**: add the value of memory location x to the content of the accumulator
 - The result stays in the accumulator
- **BR L**: branch to label L (goto)
- **BZ L**: if accumulator is zero, branch to label L
- **L: NOP**: label L, associated with a "no-operation" instruction NOP

68

Code Generation Strategy

- Synthesized attribute **code**
 - Contains a sequence of instructions
- Example:



69

Code Generation Strategy

- $\langle c \rangle_1 ::= \text{if } \langle be \rangle \text{ then } \langle c \rangle_2 \text{ else } \langle c \rangle_3$
 $\langle c \rangle_1.\text{code} := <$
 $\langle be \rangle.\text{code},$
 " BZ L1",
 $\langle c \rangle_2.\text{code},$
 " BR L2",
 " L1: NOP",
 $\langle c \rangle_3.\text{code},$
 " L2: NOP"
 $>$

70

Problems

- T1 cannot be used in $\langle ae \rangle_3.\text{code}$
 - Need to generate temporary names
- Labels L1 and L2 can't be used in $\langle c \rangle_2.\text{code}$, $\langle c \rangle_3.\text{code}$, or elsewhere
 - Need to generate label names
- Keep counter for temporary names
 - inherited **temp**
- Keep "global" counter for label names
 - inherited **labin**, synthesized **labout**

71

Code Generation for Statements

- $\langle prog \rangle ::= \langle c \rangle$
 $\langle prog \rangle.\text{code} := \langle c \rangle.\text{code}$
 $\langle c \rangle.\text{labin} := 0$
- $\langle c \rangle_1 ::= \langle c \rangle_2 ; \langle c \rangle_3$
 $\langle c \rangle_1.\text{code} :=$
 append($\langle c \rangle_2.\text{code}, \langle c \rangle_3.\text{code}$)
 $\langle c \rangle_2.\text{labin} := \langle c \rangle_1.\text{labin}$
 $\langle c \rangle_3.\text{labin} := \langle c \rangle_2.\text{labout}$
 $\langle c \rangle_1.\text{labout} := \langle c \rangle_3.\text{labout}$

72

Code Generation for Statements

```
<c> ::= skip
    <c>.code := <"NOP" >
    <c>.labout := <c>.labin
| <assign>
    <c>.code := <assign>.code
    <c>.labout := <c>.labin
```

73

Code Generation for Statements

```
<c>1 ::= if <be> then <c>2 else <c>3
    <c>2.labin := <c>1.labin + 2
    <c>3.labin := <c>2.labout
    <c>1.labout := <c>3.labout
    <c>1.code := append( <be>.code,
        ( "BZ" label(<c>1.labin) ),
        <c>2.code,
        ( "BR" label(<c>1.labin + 1) ),
        ( label(<c>1.labin) "NOP" ),
        <c>3.code,
        ( label(<c>1.labin+1) "NOP" ) )
```

74

Code Generation for Statements

```
<c>1 ::= while <be> do <c>2
    <c>2.labin := <c>1.labin + 2
    <c>1.labout := <c>2.labout
    <c>1.code := append(
        ( label(<c>1.labin) "NOP" ),
        <be>.code,
        ( "BZ" label(<c>1.labin + 1) ),
        <c>2.code,
        ( "BR" label(<c>1.labin) ),
        ( label(<c>1.labin+1) "NOP" ) )
```

75

Code Generation for Statements

```
<assign> ::= <id> := <ae>
    <ae>.temp := 1
    <assign>.code := append(
        <ae>.code,
        ("STO" <id>.name))
```

- For recursive languages, we need to access variables on the stack or heap

76

Code Generation for Expressions

```
<ae>1 ::= <int>
    <ae>1.code := < ("LOAD" <int>.value) >
    | <id>
    <ae>1.code := < ("LOAD" <id>.name) >
    | <ae>2 + <ae>3
    <ae>2.temp := <ae>1.temp
    <ae>3.temp := <ae>1.temp + 1
    <ae>1.code := append(
        <ae>2.code,
        ("STO" temp(<ae>1.temp) ),
        <ae>3.code,
        ("ADD" temp(<ae>1.temp) ) )
```

77

Example for Code Generation

- Source program
if x = 42 then
 if a = b then
 y := 1
 else
 y := 2
else
 y := 3

78

Example for Code Generation

- Generated code for outer if statement

```
# code for (x=42)
BZ L1
# code for inner if
BR L2
L1: NOP
# code for y := 3
L2: NOP
```

79

Inner If Statement

```
# code for x = 42
BZ L1
# code for a = b
BZ L3
# code for y := 1
BR L4
L3: NOP
# code for y := 2
L4: NOP
BR L2
L1: NOP
# code for y := 3
L2: NOP
```

80

Code for Assignments

```
# code for x = 42
BZ L1
# code for a = b
BZ L3
LOAD 1
STO y
BR L4
L3: NOP
LOAD 2
STO y
L4: NOP
BR L2
L1: NOP
LOAD 3
STO y
L2: NOP
```

81

Summary: Attribute Grammars

- Tool for specifying non-context-free languages
- Useful for expressing arbitrary tree walks over context-free parse trees
- Synthesized and inherited attributes
- Conditions to reject invalid parse trees
- Evaluation order depends on attribute dependencies

82

Summary: Attribute Grammar

- Realistic applications: for **type checking** and **code generation**
- "Global" data structures must be passed around as attributes
- "Global" counters (e.g. for labels) are implemented as pair of attributes
- Any data structure (sets, etc.) can be used as long as we work on paper
- The rules can call auxiliary functions

83
