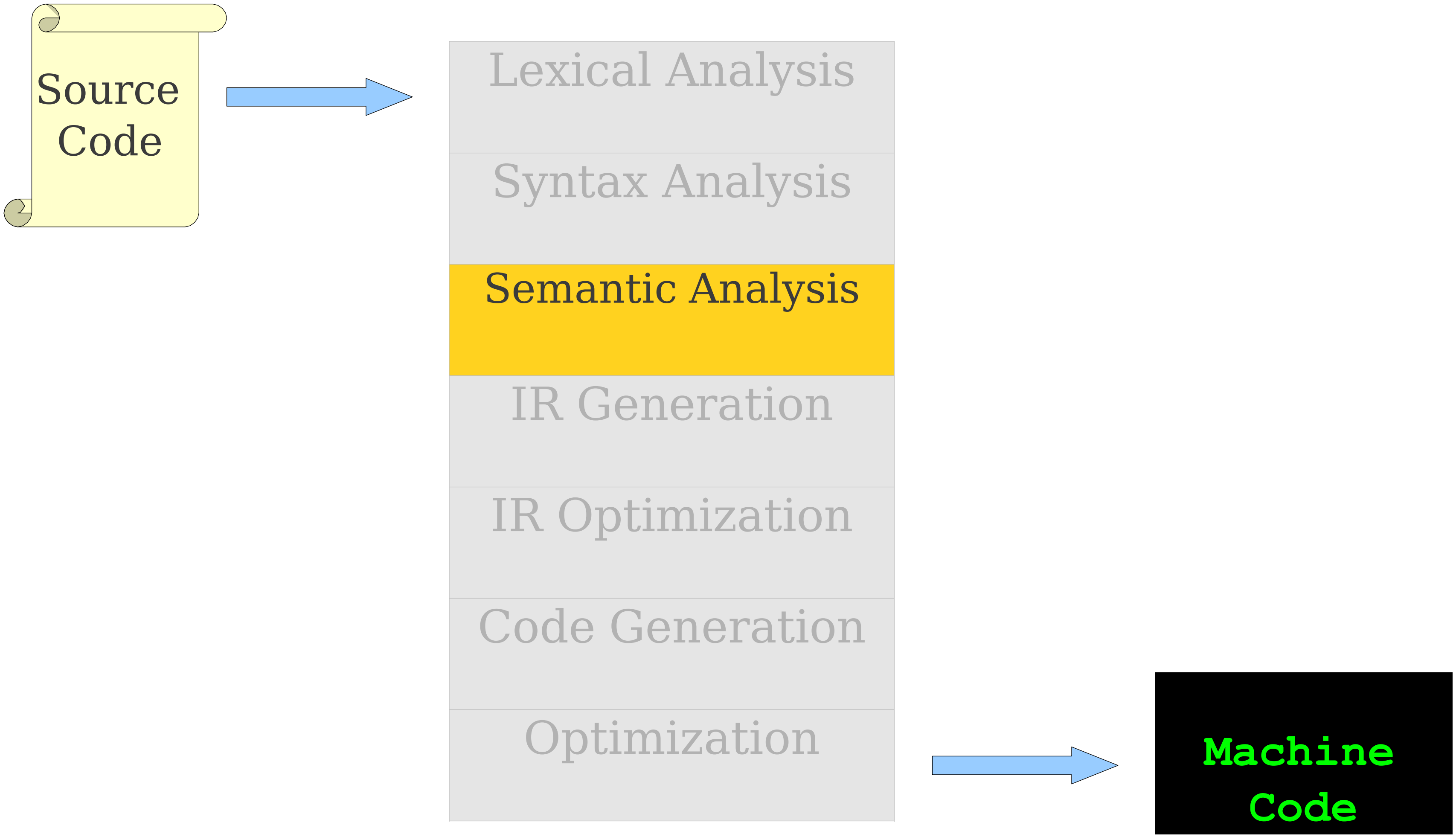
بسم الله الرحمن الرحيم

Semantic Analysis

Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Where We Are

- Program is *lexically* well-formed:
  - Identifiers have valid names.
  - Strings are properly terminated.
  - No stray characters.
- Program is *syntactically* well-formed:
  - Class declarations have the correct structure.
  - Expressions are syntactically valid.
- Does this mean that the program is legal?

A Short Decaf Program

```
class MyClass implements MyInterface
    {  string myInteger;

    void doSomething()
        {  int[] x = new
        string;

         x[5] = myInteger * y;
    }
    void doSomething() {

    }
    int fibonacci(int n) {
         return doSomething() + fibonacci(n - 1);
    }
}
```

A Short Decaf Program

```
class MyClass implements MyInterface    {
      string myInteger;

      void doSomething() {
            int[] x = new string;

            x[5] = myInteger * y;
      }
      void doSomething()     {

      }
      int fibonacci(int n) {
            return doSomething() + fibonacci(n - 1);
      }
}
```

Interface not declared

Wrong type

Can't multiply strings

Variable not declared

Can't redefine functions

Can't add void

No main function

Semantic Analysis

- Ensure that the program has a well-defined **meaning**.
- Verify properties of the program that aren't caught during the earlier phases:
  - Variables are declared before they're used.
  - Expressions have the right types.
  - Arrays can only be instantiated with **`NewArray`**.
  - Classes don't inherit from nonexistent base classes
  - ...
- Once we finish semantic analysis, we know that the user's input program is legal.

Challenges in Semantic Analysis

- Reject the largest number of incorrect programs.
- Accept the largest number of correct programs.
- Do so quickly.

Why can't we just do this during parsing?

Limitations of CFGs

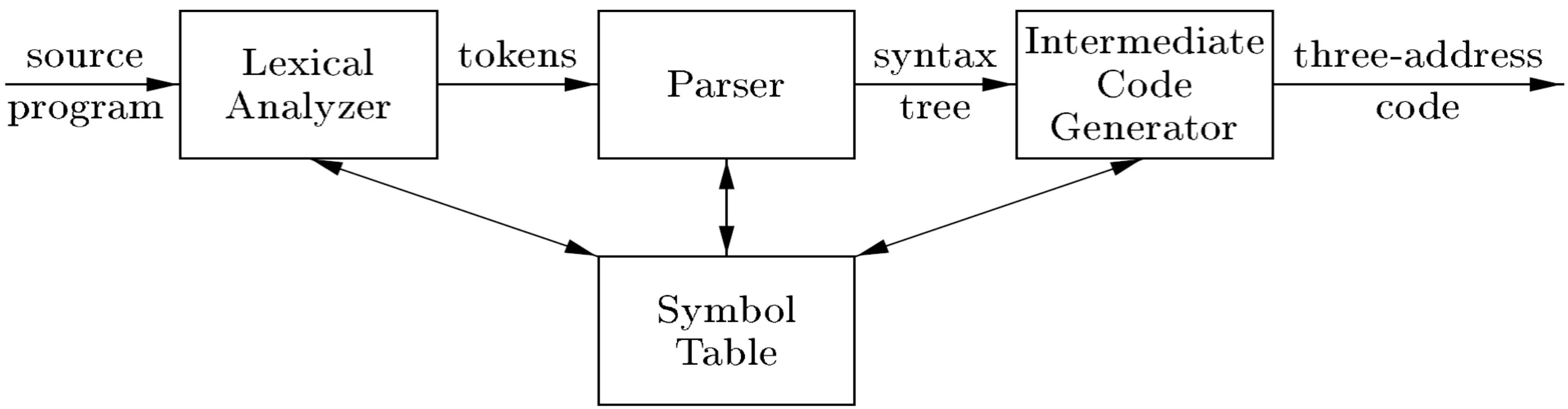
- Using CFGs:
  - How would you prevent duplicate class definitions?
  - How would you differentiate variables of one type from variables of another type?
  - How would you ensure classes implement all interface methods?
- For most programming languages, these are *provably impossible*.
  - Use the pumping lemma for context-free languages, or Ogden's lemma.

Implementing Semantic Analysis

- **Attribute Grammars**
  - Augment **`bison`** rules to do checking during parsing.
  - Approach suggested in the *Compilers* book.
  - Has its limitations; more on that later.
- **Recursive AST Walk**
  - Construct the AST, then use virtual functions and recursion to explore the tree.

A Remaining Question:

- How Does Parser and Semantic Analyzer interact?
- Recall the overall structure:



Communication to SA and CG

- Sometimes we can merge Semantic Analyzer and Code-Generating units.
- However we must design a way to communicate.
- i.e. What should we do when (e.g.) an addition is detected in program?

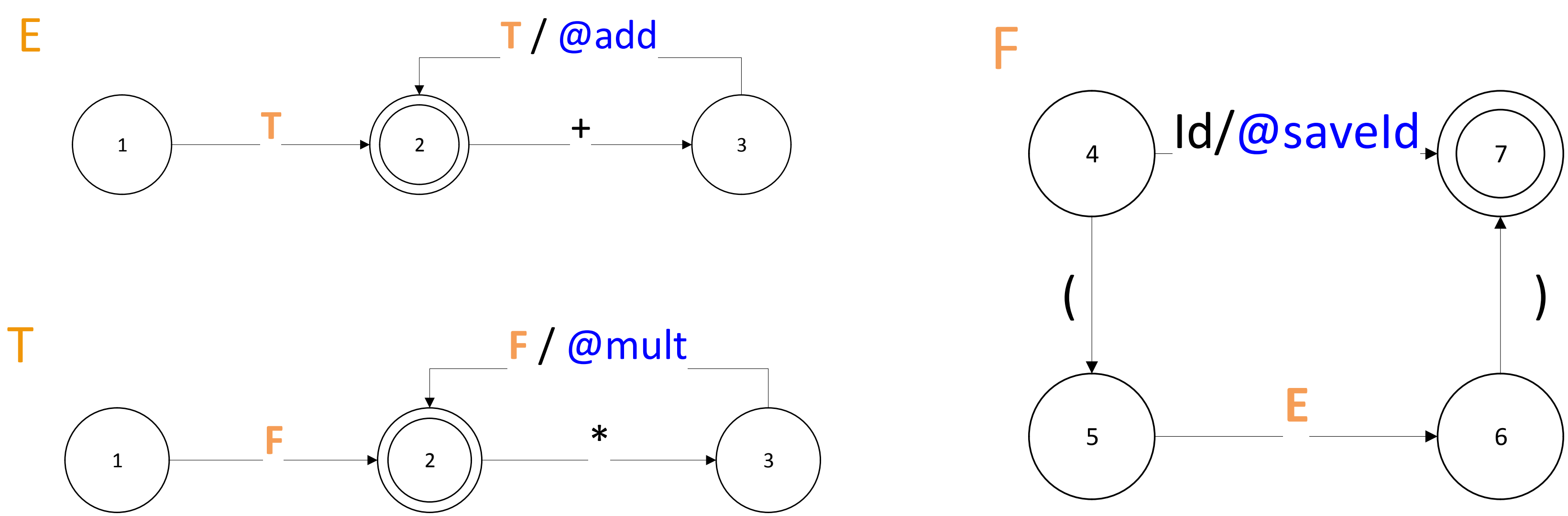
- Any idea?

Syntax Graph

- On edges, we add semantic actions if necessary:

Syntax Graph

- On edges, we add semantic actions if necessary:
- Consider the graph of expression contains add.



LR-Parser

- Each production means a piece of semantic!
- So, By each reduction we need to communicate to Semantic Analyzer or perhaps code-generator.

1. **S** → **E**
2. **E** → **E** `+` **E**
3. **E** → **E** `*` **E**
4. **E** → `(`**E**`)`
5. **E** → `int`

LL(1) and RD Parsers

- We use semantic actions inside grammar rules, as pointers to routines:

1. E → T E’
2. E’ → + T E’
3. | ε
4. T → F T’
5. T’ → * F T’
6. | ε
7. F → id

LL(1) and RD Parsers

- We use semantic actions inside grammar rules, as pointers to routines:
- Obviously, in RD Parser you can call them when you need.

1. E → T E’
2. E’ → + T **@add** E’
3. | ε
4. T → F T’
5. T’ → * F @mult T’
6. | ε
7. F → @save id

id * id

1. E → T E’
2. E’ → + T **@add** E’
3. | ε
4. T → F T’
5. T’ → * F @mult T’
6. | ε
7. F → @save id

						*
			@save			F
		F	Id	Id		@mult
	T	T’	T’	T’	T’	T’
E	E’	E’	E’	E’	E’	E’
$	$	$	$	$	$	$

Syntax-Directed Translation

What is SDT?

- *Syntax-directed translation* refers to a method of compiler implementation where we use **parse tree** to direct **semantic analysis**.
- It can be a separated phase or do along parsing.

What is SDT?

- *Syntax-directed translation* refers to a method of compiler implementation where we use **parse tree** to direct **semantic analysis**.
- It can be a separated phase or do along parsing.
- This method can be used for IR-Generation, Type-checking and also implementing small languages (hope we can talk about it more later!).

What is SDT?

- *Syntax-directed translation* refers to a method of compiler implementation where we use **parse tree** to direct **semantic analysis**.
- It can be a separated phase or do along parsing.
- This method can be used for IR-Generation, Type-checking and also implementing small languages (hope we can talk about it more later!).
- So we need an **SDD (?)**.

SDT: example

- Consider the expr:

$$expr \rightarrow expr_1 + term$$

- The translation is:
    1. `Translate expr`$_1$
    2. `Translate term`
    3. `Handle +`

- So, in order to make an expr, we should translate first expr and then the term.
- Then we can handle the addition.

Syntax Directed Definition

Syntax **D**irected **D**efinition

- We augment the grammar with information (rules and attributes), which helps us in semantic analysis.

Syntax Directed Definition

- We augment the grammar with information (rules and attributes), which helps us in semantic analysis.
- This is done via non-terminals.

Syntax **D**irected **D**efinition

- We augment the grammar with information (rules and attributes), which helps us in semantic analysis.
- This is done via non-terminals.
- Each of program construct (**symbols**) is associated with some quantity we call, attributes.

Syntax **D**irected **D**efinition

- We augment the grammar with information (rules and attributes), which helps us in semantic analysis.
- This is done via non-terminals.
- Each of program construct (**symbols**) is associated with some quantity we call, attributes.
- They can have a name and value: a string, a number, a type, a memory location and etc.

Syntax Directed Definition

- We augment the grammar with information (rules and attributes), which helps us in semantic analysis.
- This is done via non-terminals.
- Each of program construct (**symbols**) is associated with some quantity we call, attributes.
- They can have a name and value: a string, a number, a type, a memory location and etc.
- SDT is an SDD with explicitly specified the order of evaluation of semantic rules.

SDD: example 1

- Consider the expr:

$$expr \rightarrow expr_1 + term$$

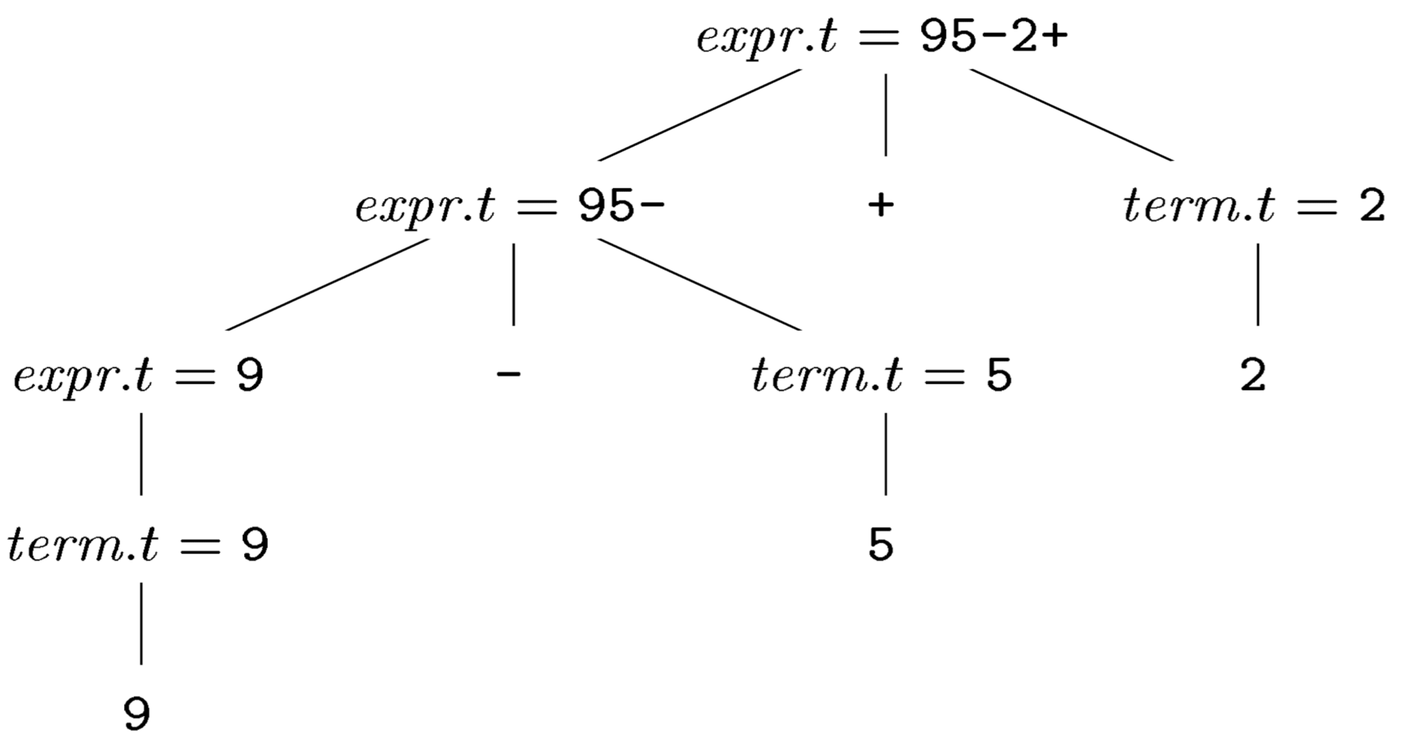
- The target is post-order so the translation is:
  1. Translate $expr_1$
  2. Translate term
  3. Handle +

- So we have the semantic rule:

$$expr.code = expr_1.code \parallel term.code \parallel +$$

SDD: example 1

- Consider single digit expr:



SDD: example 2

- Consider the following grammar:

$$INT \rightarrow INT\, DIGIT \mid DIGIT$$

$$DIGIT \rightarrow 0 \,\Big|\, 1 \ldots \Big|\, 9$$

SDD: example 2

- Consider the following grammar:

$$INT \rightarrow INT\, DIGIT \mid DIGIT$$
$$DIGIT \rightarrow 0 \,\Big|\, 1 \ldots \Big|\, 9$$

- We add attribute ‘value’ to store the correct number:

SDD: example 2

- Consider the following grammar:

$$INT \rightarrow INT\,DIGIT \mid DIGIT$$
$$DIGIT \rightarrow 0 \,\Big|\, 1 \ldots \Big|\, 9$$

- We add attribute ‘value’ to store the correct number:

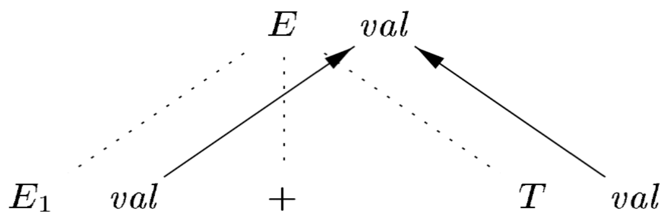
$$DIGIT \rightarrow 0\,\{DIGIT.value = 0\} \mid \ldots$$
$$INT \rightarrow DIGIT\,\{INT.value = DIGIT.value\}$$
$$INT \rightarrow \ \ldots\{INT.value = INT_1.value \ * 10 + DIGIT.value\}$$

SDD: example 2

- 435

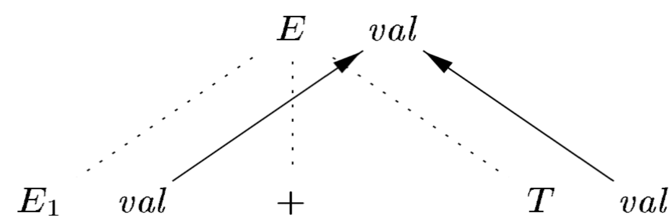
SDD vs SDT?

Dependency Graph



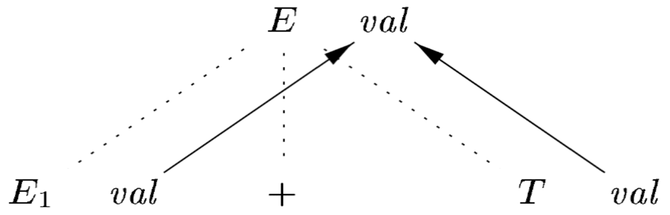
Dependency Graph

- Dependency Graph depict the flow of information among the attribute instances in a particular parse tree:



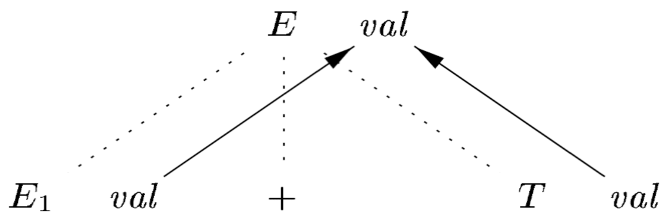
Dependency Graph

- Dependency Graph depict the flow of information among the attribute instances in a particular parse tree:
- Each node labeled with A in PT, DG has a node for each attribute associated with A.

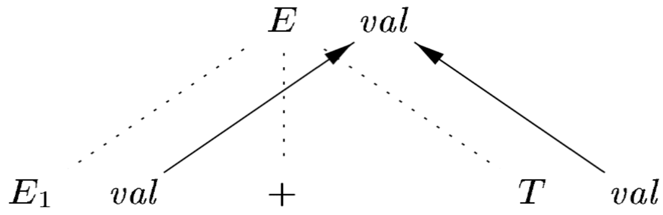


Dependency Graph

- Dependency Graph depict the flow of information among the attribute instances in a particular parse tree:
- Each node labeled with A in PT, DG has a node for each attribute associated with A.
- In a production p if $A.b = f(X.c)$ then we have a directed edge in DG from $X.c$ to $A.b$.

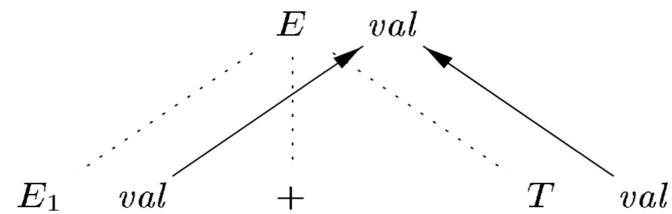


Dependency Graph



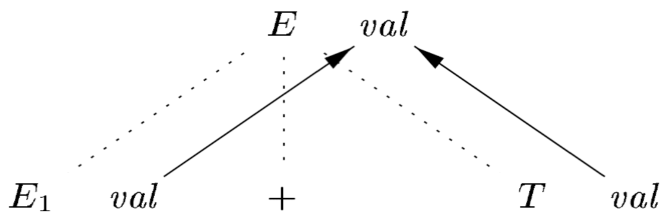
Dependency Graph

- Dependency Graph depict the flow of information among the attribute instances in a particular parse tree:



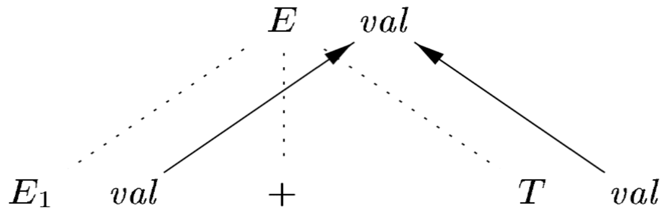
Dependency Graph

- Dependency Graph depict the flow of information among the attribute instances in a particular parse tree:
- Each node labeled with A in PT, DG has a node for each attribute associated with A.



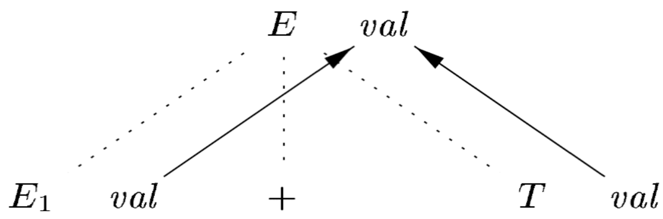
Dependency Graph

- Dependency Graph depict the flow of information among the attribute instances in a particular parse tree:
- Each node labeled with A in PT, DG has a node for each attribute associated with A.
- In a production p if $A.b = f(X.c)$ then we have a directed edge in DG from $X.c$ to $A.b$.



Dependency Graph

- Dependency Graph depict the flow of information among the attribute instances in a particular parse tree:
- Each node labeled with A in PT, DG has a node for each attribute associated with A.
- In a production p if $A.b = f(X.c)$ then we have a directed edge in DG from $X.c$ to $A.b$.
- In General topological sorting the DG give us an order for evaluation.

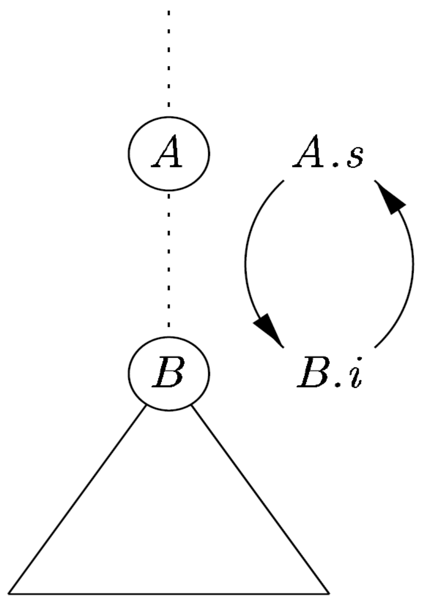


A Problem

- However, without any restriction on attribute’s code, some times it is not possible.

A Problem

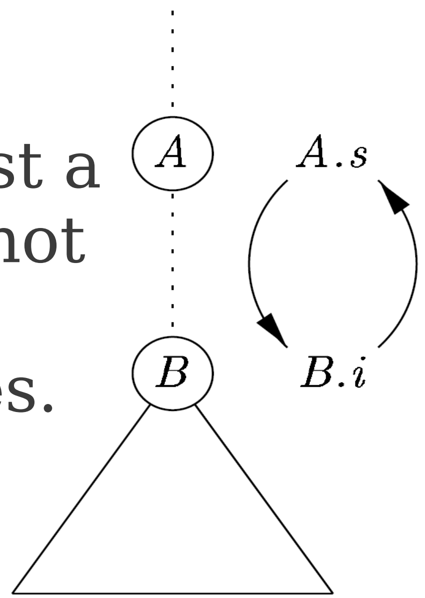
- However, without any restriction on attribute’s code, some times it is not possible.



A Problem

- However, without any restriction on attribute's code, some times it is not possible.

- In fact, check whether there exist a parse tree which has a cycle or not is *hard*.
- So we classify useful SDD classes.



Classify Attributes

- A **synthesized attribute** for a non-terminal at a node in PT, is defined only in terms of attribute values of its **descendent** in PT.
  - i.e. at node N which contains $A \rightarrow X_1 \ldots X_n$, $A.s$ relies just on it's children or itself.
- An **inherited attribute** for a nonterminal B at a PT node is defined by a semantic rule associated with the production at its parent.
  - It must have B as a symbol in its body.
  - An inherited attribute at node N is defined only in terms of attribute values at N's **parent**, N **itself**, and N's **siblings**

Classify SDDs

- **S-Attributed**: An SDD is s-attributed every attribute is synthesized.
  - They can be evaluated in any bottom-up order.
  - Can be implemented during bottom-up parsing.

Classify SDDs

- **S-Attributed**: An SDD is s-attributed every attribute is synthesized.
  - They can be evaluated in any bottom-up order.
  - Can be implemented during bottom-up parsing.
- **L-Attributed**: each attribute is either:
  - Synthesized
  - Inherited but, in $A \rightarrow X_1 \ldots X_{i-1} X_i \ldots X_n$, each $X_i . inh$ may use:
    - Only inherited attributes associated with A
    - Inherited or synthesized attributes of X_j, $j < i$
    - X_i itself but without making any loop.

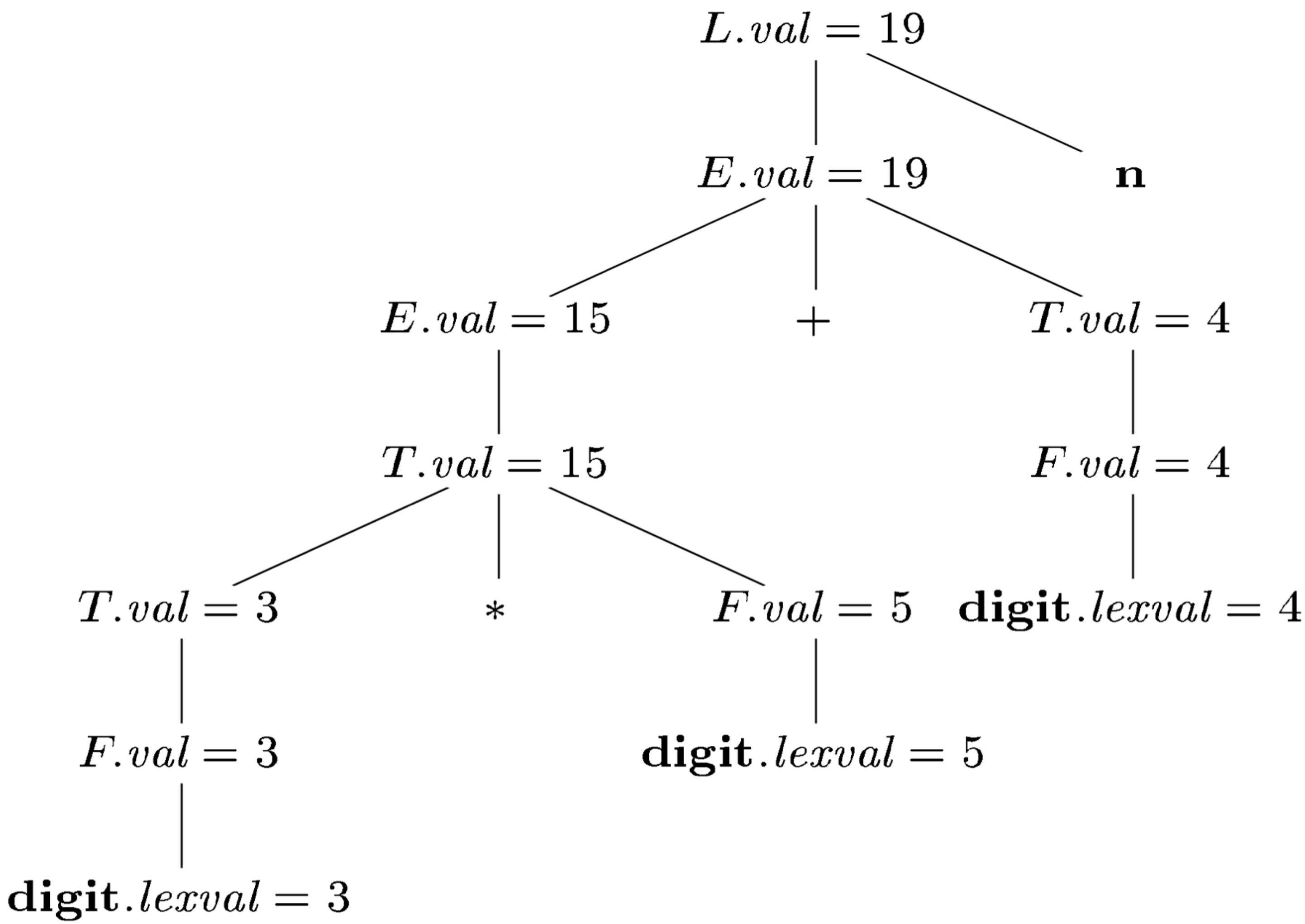
Example S-Attributed

- Consider the following SDD for calculation expr value:

	Production	Semantic Rules
1)	$L \rightarrow E\ \mathbf{n}$	$L.val = E.val$
2)	$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \rightarrow T$	$E.val = T.val$
4)	$T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \rightarrow F$	$T.val = F.val$
6)	$F \rightarrow (E)$	$F.val = E.val$
7)	$F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}.\text{lexval}$

Annotated Pares Tree

3 * 5 + 4

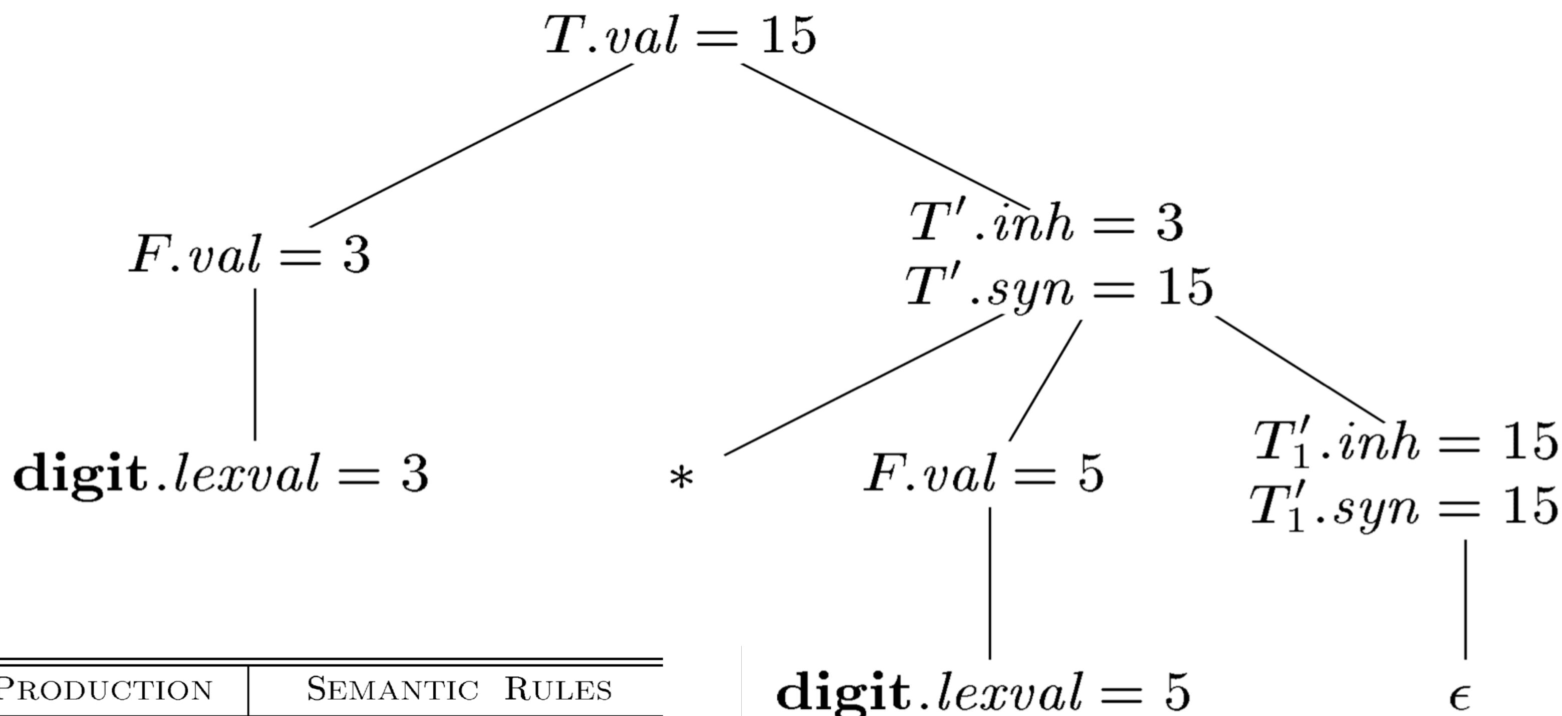


Example: L-Aattributed

- Again, expr.
- Note the difference:

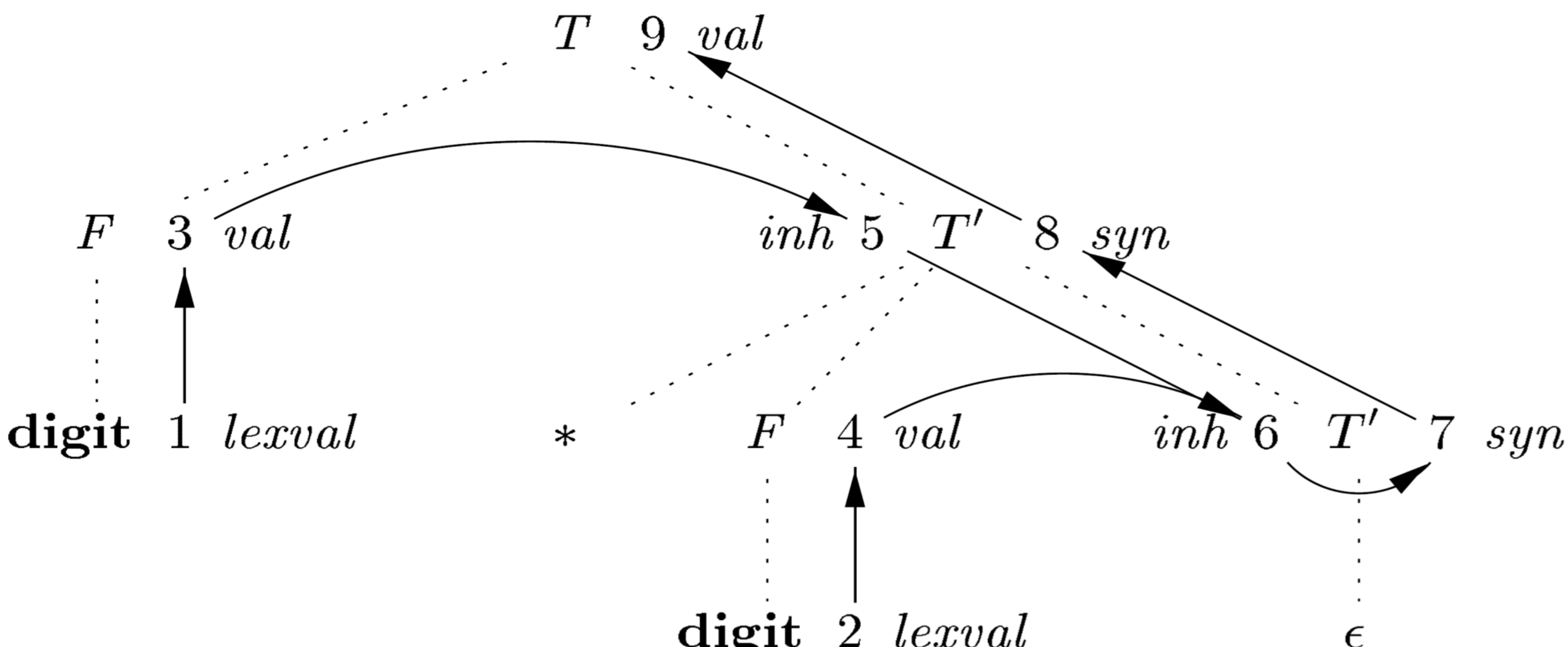
	Production	Semantic Rules
1)	$T \rightarrow F\, T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow *\, F\, T_1'$	$T_1'.inh = T'.inh \times F.val$ $T'.syn = T_1'.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$

Annotated Parse Tree



	Production	Semantic Rules
1)	$T \to F\, T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \to *\, F\, T_1'$	$T_1'.inh = T'.inh \times F.val$ $T'.syn = T_1'.syn$
3)	$T' \to \epsilon$	$T'.syn = T'.inh$
4)	$F \to \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$

Example: Dependency Graph

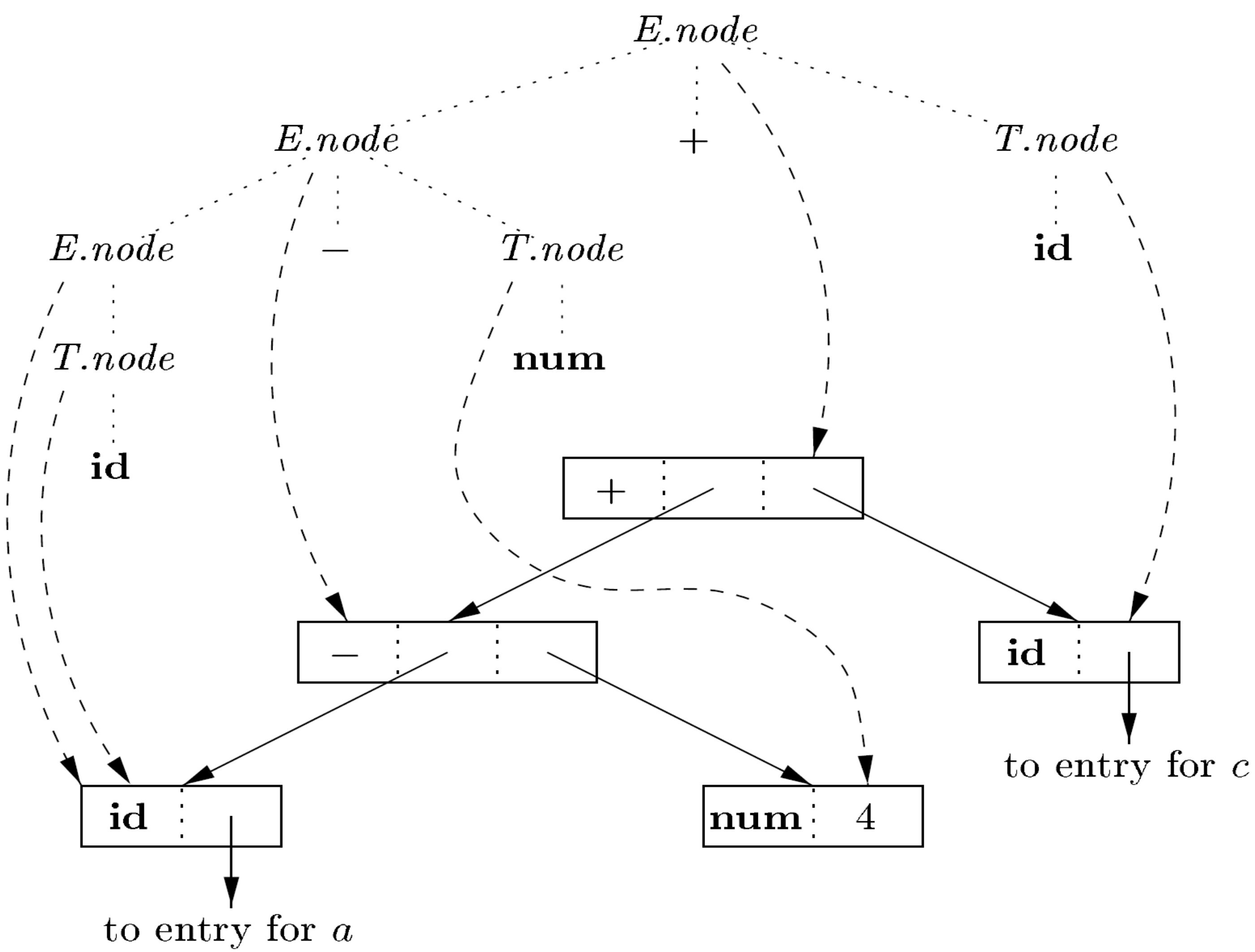


Application: Building Syntax Tree

- The following S-attributed definition construct syntax tree for simple expr grammars:

	Production	Semantic Rules
1)	$E \rightarrow E_1 + T$	$E.node = \textbf{new}\ Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node = \textbf{new}\ Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (\ E\ )$	$T.node = E.node$
5)	$T \rightarrow \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \rightarrow \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

Syntax Tree



E.node
E.node
+
T.node
E.node
−
T.node
id
T.node
num
id
+
−
id
to entry for c
id
num
4
to entry for a

How to Implement SDD?

- We use Syntax-Directed Translation Schema to translate SDD.
- To do so:
  - Find parse tree without code fragments.
  - Then add code fragments.
  - At the end, by traverse the tree and run the code complete the translation.
- Typically, SDT's are implemented during parsing, without building a parse tree.

Syntax-Directed Translation Schemes

- A translation scheme is a notation for attaching *program fragments* to the productions of a grammar.

Syntax-Directed Translation Schemes

- A translation scheme is a notation for attaching *program fragments* to the productions of a grammar.
- The program fragments are **executed** when the production is used during syntax analysis.

Syntax-Directed Translation Schemes

- A translation scheme is a notation for attaching *program fragments* to the productions of a grammar.
- The program fragments are **executed** when the production is used during syntax analysis.
- These grammar embedded fragments calls *semantic actions*!

Syntax-Directed Translation Schemes

- A translation scheme is a notation for attaching *program fragments* to the productions of a grammar.
- The program fragments are **executed** when the production is used during syntax analysis.
- These grammar embedded fragments calls *semantic actions*!
- The combined result of all these fragment executions, produces the translation of the program.

Example: LR-Parser

- It can be implemented using a stack.
- The attribute(s) of each grammar symbol can be put on the stack in a so they can be found during the reduction.

$$
\begin{array}{lcll}
L & \rightarrow & E \ \mathbf{n} & \{ \ \mathrm{print}(E.val); \ \} \\
E & \rightarrow & E_1 + T & \{ \ E.val = E_1.val + T.val; \ \} \\
E & \rightarrow & T & \{ \ E.val = T.val; \ \} \\
T & \rightarrow & T_1 * F & \{ \ T.val = T_1.val \times F.val; \ \} \\
T & \rightarrow & F & \{ \ T.val = F.val; \ \} \\
F & \rightarrow & (\ E \ ) & \{ \ F.val = E.val; \ \} \\
F & \rightarrow & \mathbf{digit} & \{ \ F.val = \mathbf{digit}.lexval; \ \}
\end{array}
$$

SDD with Action Inside Production

- Consider productions like:

$A \; \rightarrow X \, \{a\} \, Y$

SDD with Action Inside Production

- Consider productions like:

$A \rightarrow X \{a\} Y$

- In LR-parsers action a perform after find handle X or shift X.

SDD with Action Inside Production

- Consider productions like:

$A \rightarrow X \{a\} Y$

- In LR-parsers action a perform after find handle X or shift X.
- In Top-Down parsers, action a before expand Y or check Y on input.

SDD 2 SDT

- The semantic rules in an SDD can be converted into an SDT with actions that are executed at the right time.
- During parsing, an action in a production body is executed as soon as all the grammar symbols to the left of the action have been matched.
- What should we do with middle actions?
- For each middle action we add a *distinct* **marker nonterminal.** E.g. M_1. It has just one production $M_1 \to \epsilon$.

It can be problematic…

- Infix to prefix conversion.
- Using markers M_2 and M_4 for productions 2 and 4 respectively.
- A **digit** token face R/R conflict! (LR)
- Or print operator before they appear!

$$
\begin{array}{llll}
1) & L & \rightarrow & E\ \mathbf{n} \\
2) & E & \rightarrow & \{\ \mathrm{print}('+');\ \}\ \ E_1 + T \\
3) & E & \rightarrow & T \\
4) & T & \rightarrow & \{\ \mathrm{print}('*');\ \}\ \ T_1 * F \\
5) & T & \rightarrow & F \\
6) & F & \rightarrow & (\ E\ ) \\
7) & F & \rightarrow & \mathbf{digit}\ \ \{\ \mathrm{print}(\mathbf{digit}.lexval);\ \}
\end{array}
$$

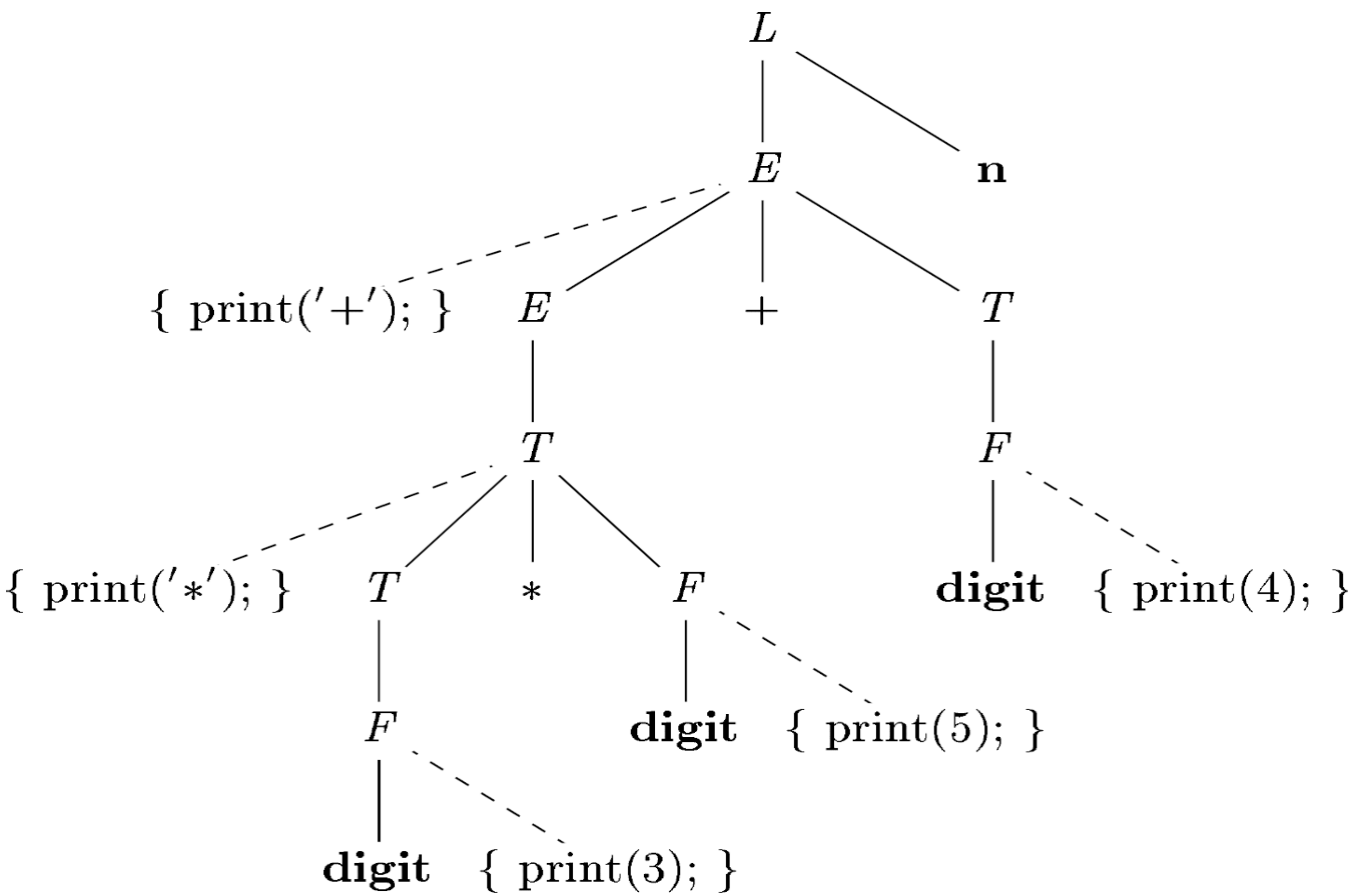
A General Solution

1. Ignoring the actions, parse the input and produce a parse tree as a result.

2. For each interior node N, production $A \rightarrow \alpha$: add additional children to N for the actions in α, so the children of N from left to right have exactly the symbols and actions of α.

3. Perform a **preorder** traversal, as soon as a node with action visited, perform the action.

Example



L
E
n
{ print('+'); }
E
+
T
T
F
{ print('*'); }
T
*
F
digit
{ print(4); }
F
digit
{ print(5); }
digit
{ print(3); }

SDT for L-Attributed

- The Rule is as follow:
- Embed the action that computes the **inherited attributes** for a nonterminal A immediately before that occurrence of A in the body of the production in order that those needed first are computed first.
- Place the actions that compute a synthesized attribute for the head of a production at the end of the body of that production

Example

- Loop:

$S \rightarrow \boldsymbol{while}\ (C)\ S_1$

- Here, S is a nonterminal generates all kinds of statements. C here is conditional statement.
- We use the following attributes:
- $S.next$: beginning of the code after S.
- $S.code$: code of loop body with jump at end.
- $C.true$: beginning of the code that must be executed if C is true.
- $C.false$: labels the beginning of the code that must be executed if C is false.
- $C.code$: the code of C with appropriate jumps.

Example: L-Attributed SDD

$$S \rightarrow \textbf{while}\ (\ C\ )\ S_1 \qquad \begin{array}{l} L1 = new(); \\ L2 = new(); \\ S_1.next = L1; \\ C.false = S.next; \\ C.true = L2; \\ S.code = \textbf{label} \parallel L1 \parallel C.code \parallel \textbf{label} \parallel L2 \parallel S_1.code \end{array}$$

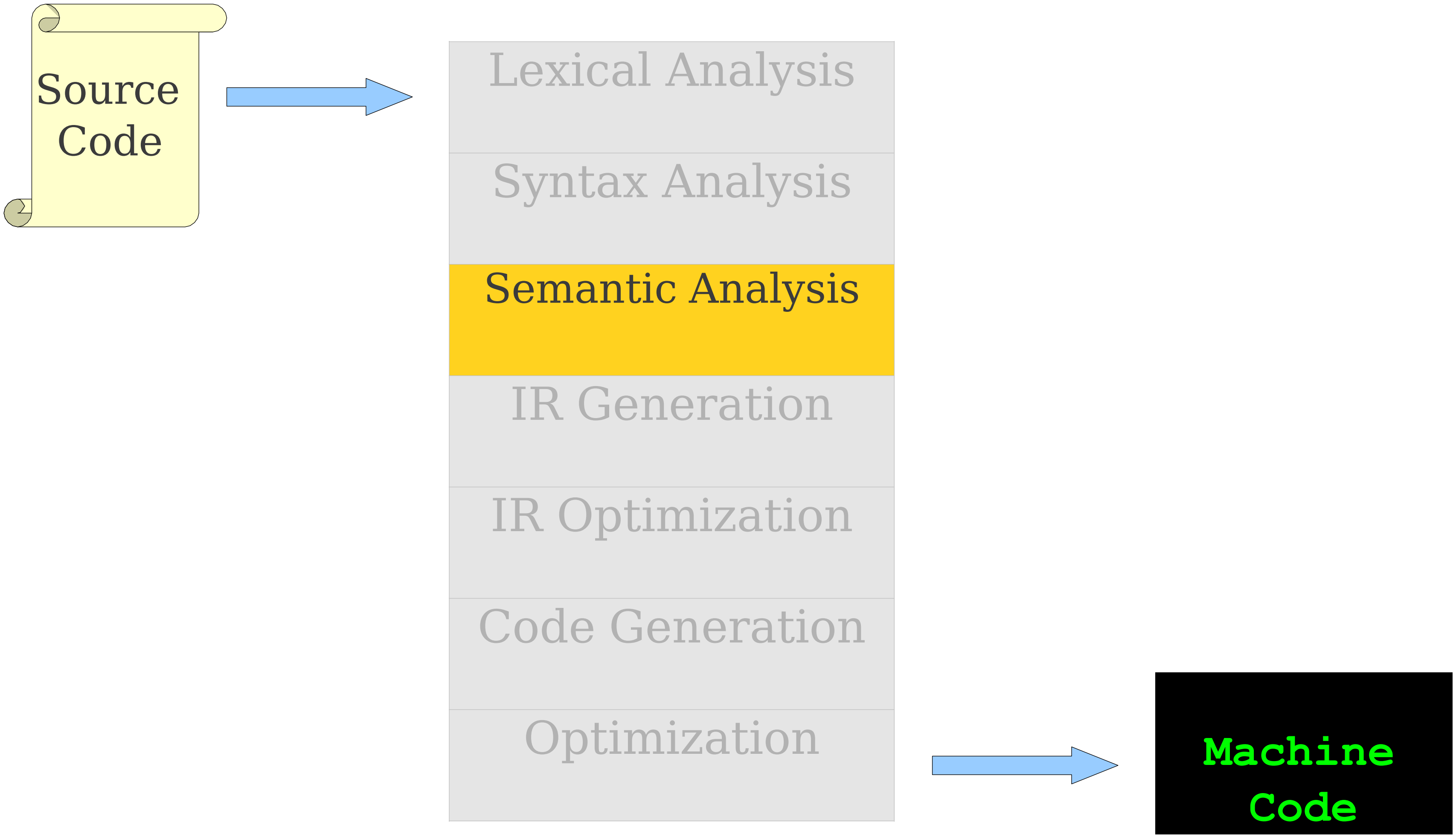
Example: L-Attributed SDD

$$
\begin{array}{ll}
S \rightarrow \textbf{while}\ (\ C\ )\ S_1 & L1 = new(); \\
& L2 = new(); \\
& S_1.next = L1; \\
& C.false = S.next; \\
& C.true = L2; \\
& S.code = \textbf{label} \parallel L1 \parallel C.code \parallel \textbf{label} \parallel L2 \parallel S_1.code
\end{array}
$$

$$
\begin{array}{ll}
S \rightarrow \textbf{while}\ (& \{\ L1 = new();\ L2 = new();\ C.false = S.next;\ C.true = L2;\ \} \\
\qquad C\ ) & \{\ S_1.next = L1;\ \} \\
\qquad S_1 & \{\ S.code = \textbf{label} \parallel L1 \parallel C.code \parallel \textbf{label} \parallel L2 \parallel S_1.code;\ \}
\end{array}
$$

Scope

Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

What's in a Name?

- The same name in a program may refer to fundamentally different things:
- This is perfectly legal Java code:

```
public class A {
    char A;
    A A(A A) {
        A.A = 'A';
        return A((A)
        A);
    }
}
```

What's in a Name?

- The same name in a program may refer to completely different objects:
- This is perfectly legal C++ code:

```
int Awful() {
    int x = 137;
    {
        string x = "Scope!"
        if (float x = 0)
            double x = x;
    }
    if (x == 137) cout << "Y";
}
```

What's in a Name?

- The same name in a program may refer to completely different objects:
- This is perfectly legal C++ code:

```
int Awful() {
    int x = 137;
    {
        string x = "Scope!"
        if (float x = 0)
            double x = x;
    }
    if (x == 137) cout << "Y";
}
```