50.051 Programming Language Concepts

W10-S1 From Context-Free Grammars to Syntax-Directed Translation

Matthieu De Mari

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Introducing Context Free Grammars

Definition (Context-Free Grammars):

A Context-Free Grammar (CFG) is **a formal system used to generate and describe sets of strings based on a specific set of syntax rules**.

It is particularly useful for defining the syntax of programming languages and the structure of natural languages.

The term "context-free" means that the **production rules** of the CFG are applied independently of the surrounding context.

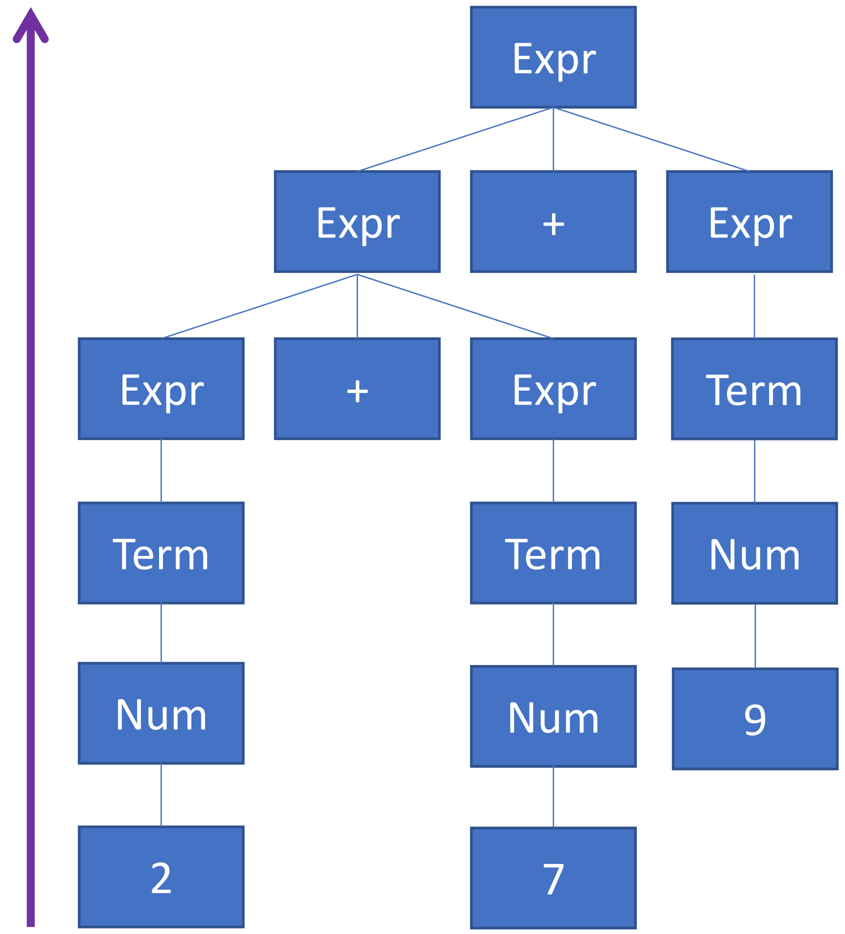
A context free grammar is defined by **four elements**: a set of **terminals** and **non-terminals**, a **start symbol** and a set of **production rules**.

Parse tree of a derivation

Derivation (parse tree of a CFG derivation):

For a given CFG derivation, we can build a parse tree,

- Whose root is the start symbol,
- Where every production rule, $X \rightarrow Y_1 \ldots Y_N$ in the derivation sequence, adds children nodes $Y_1, \ldots, Y_N$ to the node X.



This parse tree is interesting because it shows the **order** in which we should compute the different operations, starting with 2 and 7, then 2+7, and finally (2+7)+9.

Ambiguity

Definition (Ambiguity in a CFG derivation):

When using a CFG to check the syntax validity of an expression and building a parse tree, we say that a **CFG is ambiguous** if it can lead to to different derivations, with two different parse trees.

In the case of arithmetic expressions and programming languages, this means that

- Two different derivations might exist,
- Producing two different parse trees,
- And the result of both operations following the two parse trees might differ and lead to different outcomes for a given program (not good!).

Checking ambiguity algorithmically

Theorem (On checking the ambiguity of a CFG algorithmically):

Let us consider a given CFG.

There is no algorithm to check if a given CFG is ambiguous or not.

This is known as the ambiguity problem for context-free grammars, and it is proven to be an undecidable problem.

*(**Note:** Similarly, there is no general algorithm that can determine whether a given program contains an infinite loop. This is known as the Halting problem, and means that you cannot define a compiler program that can check for the presence of infinite loops in the compiled source code, let alone fix them.)*

Ambiguous grammars and compilers

Ambiguous grammars can cause significant problems in compilers.

- They make it challenging to derive the intended meaning or structure of the input program.
- Moreover, they can lead to incorrect parsing, which may result in incorrect intermediate code generation or even compiler crashes.
- Therefore, it is crucial to eliminate ambiguity in context-free grammars used in compiler design.

While checking that a CFG might be ambiguous is impossible algorithmically, there are however **manual methods** for designing CFGs that will be non-ambiguous.

Ambiguous grammars and compilers

Not all context-free grammars can be made non-ambiguous.

- Most CFGs used in programming, such as the CFG for checking balanced parentheses or the CFG for arithmetic expressions, can be represented using non-ambiguous grammars (that’s a relief!).
- **But, ultimately, it is your responsibility as a programming language designer to define rules for the syntax of your programming language that can be translated into non-ambiguous CFGs!**
- Keep in mind that ambiguous grammars can often appear more concise or easier to define than their non-ambiguous counterparts. However, the problems they cause in compiler design often outweigh the benefits of simplicity or conciseness.

Ambiguous grammars and fixing CFGs

To guarantee non-ambiguity for our CFGs and ensure that there is only one correct parse tree for any given string, there are a few manual techniques we can implement.

Some techniques for resolving ambiguity in CFGs manually:

- **Rewriting the grammar entirely,**
- **Introducing precedence,**
- **Introducing delimiters,**
- **Introducing associativity rules.**

CFG for the whole language of C?!

Why did we bring CFGs into the discussion anyway?

Because the entirety of the C language syntax rules can be described using a CFG!

(That is the sign of a well-designed syntax in a given programming language!)

```
%token IDENTIFIER CONSTANT STRING_LITERAL SIZEOF
%token PTR_OP INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
%token AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
%token SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
%token XOR_ASSIGN OR_ASSIGN TYPE_NAME

%token TYPEDEF EXTERN STATIC AUTO REGISTER
%token CHAR SHORT INT LONG SIGNED UNSIGNED FLOAT DOUBLE CONST VOLATILE VOID
%token STRUCT UNION ENUM ELLIPSIS

%token CASE DEFAULT IF ELSE SWITCH WHILE DO FOR GOTO CONTINUE BREAK RETURN

%start translation_unit
%%

primary_expression
        : IDENTIFIER
        | CONSTANT
        | STRING_LITERAL
        | '(' expression ')'
        ;

postfix_expression
        : primary_expression
        | postfix_expression '[' expression ']'
        | postfix_expression '(' ')'
        | postfix_expression '(' argument_expression_list ')'
        | postfix_expression '.' IDENTIFIER
        | postfix_expression PTR_OP IDENTIFIER
        | postfix_expression INC_OP
        | postfix_expression DEC_OP
        ;

argument_expression_list
        : assignment_expression
        | argument_expression_list ',' assignment_expression
        ;

unary_expression
        : postfix_expression
        | INC_OP unary_expression
        | DEC_OP unary_expression
        | unary_operator cast_expression
        | SIZEOF unary_expression
        | SIZEOF '(' type_name ')'
        ;

unary_operator
        : '&'
        | '*'
        | '+'
        | '-'
        | '~'
        | '!'
        ;

cast_expression
        : unary_expression
        | '(' type_name ')' cast_expression
        ;
```

CFG for the whole language of C?!

Writing a CFG for the whole language of C is hard, but a much-needed task.

If you must, approach it step-by-step by

- Figuring out CFGs for expressions first (numerical, Boolean, strings, etc.),
- Figuring out CFGs block structures (if, for, while, functions, classes, etc.).

```
%token IDENTIFIER CONSTANT STRING_LITERAL SIZEOF
%token PTR_OP INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
%token AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
%token SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
%token XOR_ASSIGN OR_ASSIGN TYPE_NAME

%token TYPEDEF EXTERN STATIC AUTO REGISTER
%token CHAR SHORT INT LONG SIGNED UNSIGNED FLOAT DOUBLE CONST VOLATILE VOID
%token STRUCT UNION ENUM ELLIPSIS

%token CASE DEFAULT IF ELSE SWITCH WHILE DO FOR GOTO CONTINUE BREAK RETURN

%start translation_unit
%%

primary_expression
        : IDENTIFIER
        | CONSTANT
        | STRING_LITERAL
        | '(' expression ')'
        ;

postfix_expression
        : primary_expression
        | postfix_expression '[' expression ']'
        | postfix_expression '(' ')'
        | postfix_expression '(' argument_expression_list ')'
        | postfix_expression '.' IDENTIFIER
        | postfix_expression PTR_OP IDENTIFIER
        | postfix_expression INC_OP
        | postfix_expression DEC_OP
        ;

argument_expression_list
        : assignment_expression
        | argument_expression_list ',' assignment_expression
        ;

unary_expression
        : postfix_expression
        | INC_OP unary_expression
        | DEC_OP unary_expression
        | unary_operator cast_expression
        | SIZEOF unary_expression
        | SIZEOF '(' type_name ')'
        ;

unary_operator
        : '&'
        | '*'
        | '+'
        | '-'
        | '~'
        | '!'
        ;

cast_expression
        : unary_expression
        | '(' type_name ')' cast_expression
        ;
```

CFG for the whole language of C?!

~~Writing a CFG for the whole language of C is hard, but a much-needed task.~~

~~If you must, approach it step-by-step by~~

- ~~Figuring out CFGs for expressions first (numerical, Boolean, strings, etc.),~~
- ~~Figuring out CFGs block structures (if, for, while, functions, classes, etc.).~~

```
%token IDENTIFIER CONSTANT STRING_LITERAL SIZEOF
%token PTR_OP INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
%token AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
%token SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
%token XOR_ASSIGN OR_ASSIGN TYPE_NAME

%token TYPEDEF EXTERN STATIC AUTO REGISTER
%token CHAR SHORT INT LONG SIGNED UNSIGNED FLOAT DOUBLE CONST VOLATILE VOID
%token STRUCT UNION ENUM ELLIPSIS

%token CASE DEFAULT IF ELSE SWITCH WHILE DO FOR GOTO CONTINUE BREAK RETURN

%start translation_unit
%%

primary_expression
        : IDENTIFIER
        | CONSTANT
        | STRING_LITERAL
        | '(' expression ')'
        ;

postfix_expression
        : primary_expression
        | postfix_expression '[' expression ']'
        | postfix_expression '(' ')'
        | postfix_expression '(' argument_expression_list ')'
        | postfix_expression '.' IDENTIFIER
        | postfix_expression PTR_OP IDENTIFIER
        | postfix_expression INC_OP
        | postfix_expression DEC_OP
        ;

argument_expression_list
        : assignment_expression
        | argument_expression_list ',' assignment_expression
        ;

unary_expression
        : postfix_expression
        | INC_OP unary_expression
        | DEC_OP unary_expression
        | unary_operator cast_expression
        | SIZEOF unary_expression
        | SIZEOF '(' type_name ')'
        ;

unary_operator
        : '&'
        | '*'
        | '+'
        | '-'
        | '~'
        | '!'
        ;

cast_expression
        : unary_expression
        | '(' type_name ')' cast_expression
        ;
```

Restricted

CFG for the whole language of C?!

Damn difficult

Very happy that people have done it and we do not have to do it ourselves!

E.g., the YACC CFG, uses the notation “A : B” instead of “A → B” and “|” for “or”.

http://www.lysator.liu.se/c/ANSI-C-grammar-y.html

```
%token IDENTIFIER CONSTANT STRING_LITERAL SIZEOF
%token PTR_OP INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
%token AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
%token SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
%token XOR_ASSIGN OR_ASSIGN TYPE_NAME

%token TYPEDEF EXTERN STATIC AUTO REGISTER
%token CHAR SHORT INT LONG SIGNED UNSIGNED FLOAT DOUBLE CONST VOLATILE VOID
%token STRUCT UNION ENUM ELLIPSIS

%token CASE DEFAULT IF ELSE SWITCH WHILE DO FOR GOTO CONTINUE BREAK RETURN

%start translation_unit
%%

primary_expression
        : IDENTIFIER
        | CONSTANT
        | STRING_LITERAL
        | '(' expression ')'
        ;

postfix_expression
        : primary_expression
        | postfix_expression '[' expression ']'
        | postfix_expression '(' ')'
        | postfix_expression '(' argument_expression_list ')'
        | postfix_expression '.' IDENTIFIER
        | postfix_expression PTR_OP IDENTIFIER
        | postfix_expression INC_OP
        | postfix_expression DEC_OP
        ;

argument_expression_list
        : assignment_expression
        | argument_expression_list ',' assignment_expression
        ;

unary_expression
        : postfix_expression
        | INC_OP unary_expression
        | DEC_OP unary_expression
        | unary_operator cast_expression
        | SIZEOF unary_expression
        | SIZEOF '(' type_name ')'
        ;

unary_operator
        : '&'
        | '*'
        | '+'
        | '-'
        | '~'
        | '!'
        ;

cast_expression
        : unary_expression
        | '(' type_name ')' cast_expression
        ;
```

Restricted

CFG for the whole language of C?!

But, of course, if you plan to create your own programming language…

…Then figuring out the CFG for the syntax of your language is your job!

- **Difficult!** (so maybe do not try and make your own programming language…!)
- **But not impossible** (just requires to be very very well-organized…)

```
%token IDENTIFIER CONSTANT STRING_LITERAL SIZEOF
%token PTR_OP INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
%token AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
%token SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
%token XOR_ASSIGN OR_ASSIGN TYPE_NAME

%token TYPEDEF EXTERN STATIC AUTO REGISTER
%token CHAR SHORT INT LONG SIGNED UNSIGNED FLOAT DOUBLE CONST VOLATILE VOID
%token STRUCT UNION ENUM ELLIPSIS

%token CASE DEFAULT IF ELSE SWITCH WHILE DO FOR GOTO CONTINUE BREAK RETURN

%start translation_unit
%%

primary_expression
        : IDENTIFIER
        | CONSTANT
        | STRING_LITERAL
        | '(' expression ')'
        ;

postfix_expression
        : primary_expression
        | postfix_expression '[' expression ']'
        | postfix_expression '(' ')'
        | postfix_expression '(' argument_expression_list ')'
        | postfix_expression '.' IDENTIFIER
        | postfix_expression PTR_OP IDENTIFIER
        | postfix_expression INC_OP
        | postfix_expression DEC_OP
        ;

argument_expression_list
        : assignment_expression
        | argument_expression_list ',' assignment_expression
        ;

unary_expression
        : postfix_expression
        | INC_OP unary_expression
        | DEC_OP unary_expression
        | unary_operator cast_expression
        | SIZEOF unary_expression
        | SIZEOF '(' type_name ')'
        ;

unary_operator
        : '&'
        | '*'
        | '+'
        | '-'
        | '~'
        | '!'
        ;

cast_expression
        : unary_expression
        | '(' type_name ')' cast_expression
        ;
```

About Syntax Directed Translation

In this next part of the lecture, we will introduce **Syntax-Directed Translation (SDT)** and **Abstract Syntax Trees (AST)**.

- SDT combines the syntactic structure of a source language
- With semantic rules to generate a target language representation, such as intermediate code, abstract syntax trees, or later on the target machine code we want as the result of compilation.

In a sense, we are already looking ahead, at the next steps,

- **Trying to connect our parse trees,**
- **With the semantic analysis and the intermediate code generation.**
- *(To be discussed on Week 11!).*

Syntax Directed Translation: definitions

Definition (Syntax-Directed Translation):

Syntax-directed Translation is a **formalism used to describe the translation process in the parse trees generated by a given CFG**.

They associate

- **Semantic actions** and **attribute computations**
- With **the production rules, that have been used in a parse tree**.

This enables a systematic approach to generating target language representations from source language constructs.

Syntax-directed definitions consist of two main components, added to a CFG definition, namely **attributes** and **rules**.

Syntax Directed Translation: definitions

Definition (Attributes):

Attributes are **properties associated with grammar symbols**, for both terminals symbols (typically represented as tokens, resulting from the Tokenization) and non-terminals symbols.

These attributes **store semantic information required for the translation process**. Attributes will typically be used to:

- Hold intermediate results,
- Hold information about the structure of the source language,
- Or store information about the target language representation.

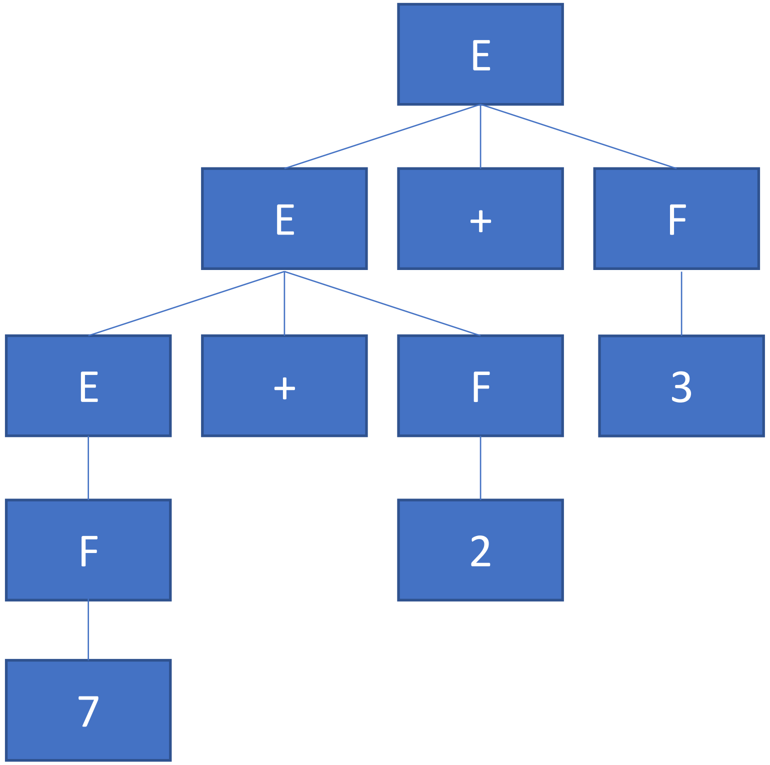
Two types of attributes: **synthesized attributes** and **inherited attributes**.

Syntax Directed Translation: definitions

Definition (Synthetized Attributes):

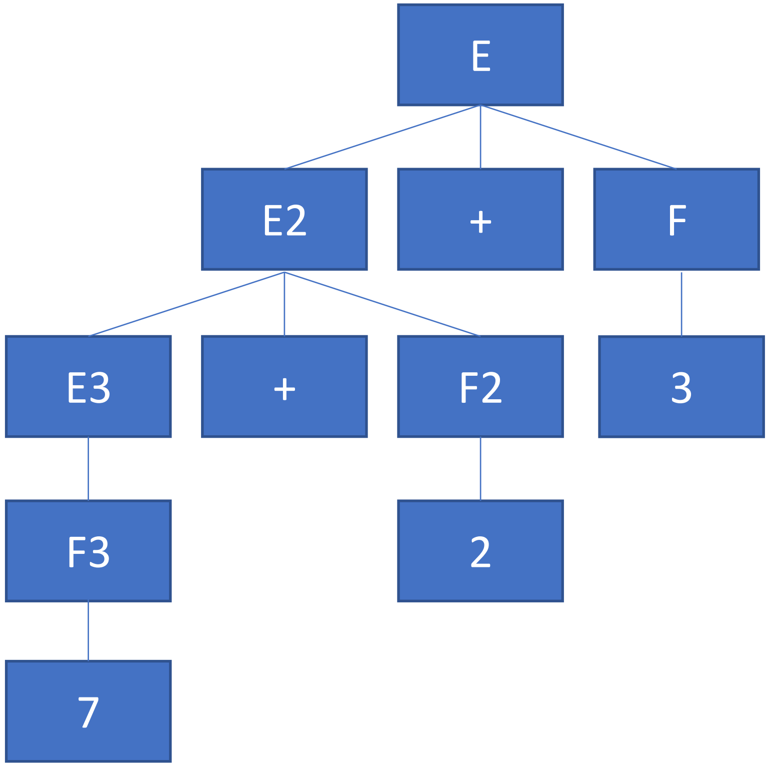
Synthesized Attributes are information contained in the children nodes in the parse tree, and are often following from the tokens descriptions.

These attributes **propagate information up the tree**, from the leaves (terminals) towards the root (start symbol).



Syntax Directed Translation: definitions

We can define a synthesized attribute 'value' for all Ex and Fx symbols.



Now…

Consider a production E2 → E3 + F2

The value of E2 would be calculated using the values of E3 and F2, as in E2.value = E3.value + F2.value.

This operation happens whenever the E → E + F CFG rule is used.

Restricted

Syntax Directed Translation: definitions

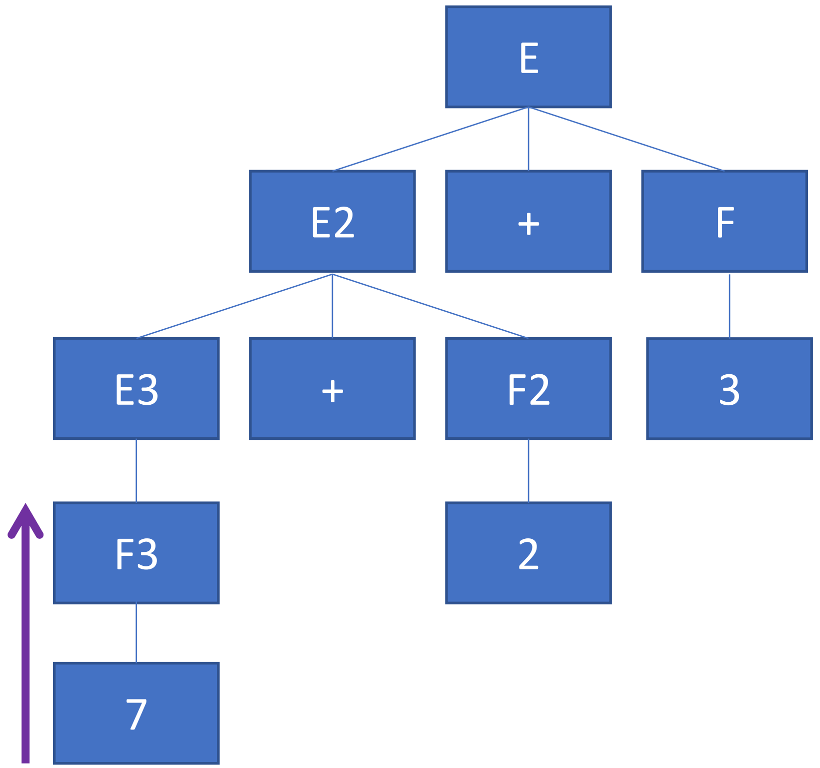
We can define a synthesized attribute 'value' for all Ex and Fx symbols.

Now…

Consider a production E2 → E3 + F2

The value of E2 would be calculated using the values of E3 and F2, as in E2.value = E3.value + F2.value.

This operation happens whenever the E → E + F CFG rule is used.



Syntax Directed Translation: definitions

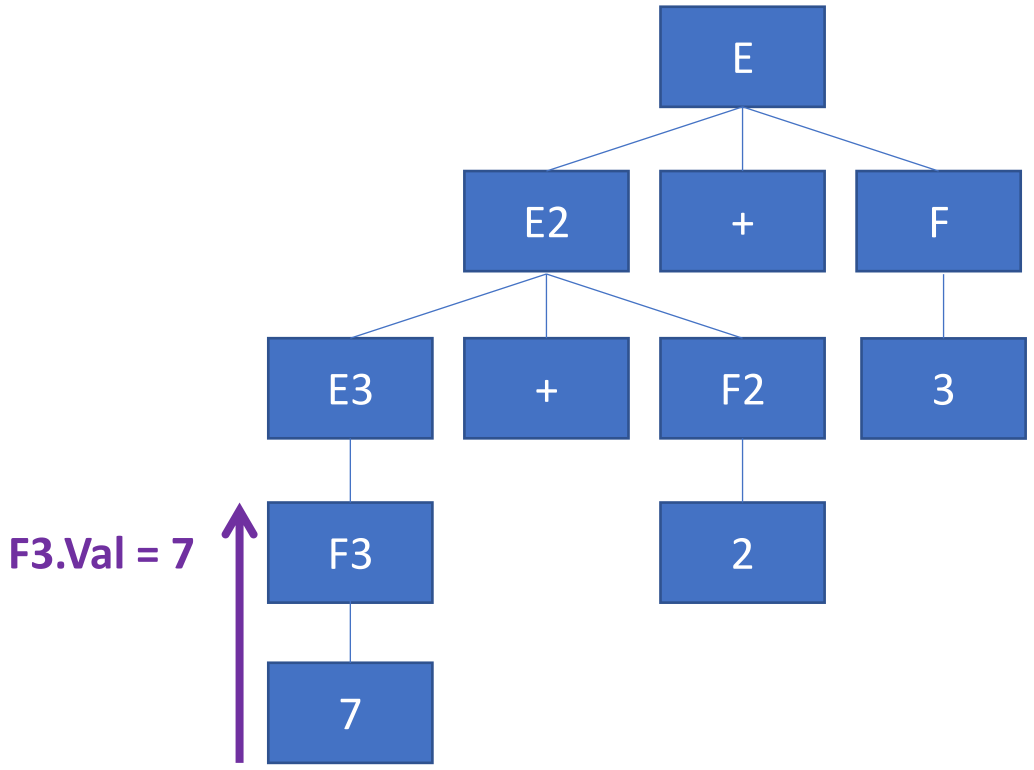
We can define a synthesized attribute 'value' for all Ex and Fx symbols.

Now…

Consider a production E2 → E3 + F2

The value of E2 would be calculated using the values of E3 and F2, as in E2.value = E3.value + F2.value.

This operation happens whenever the E → E + F CFG rule is used.



Syntax Directed Translation: definitions

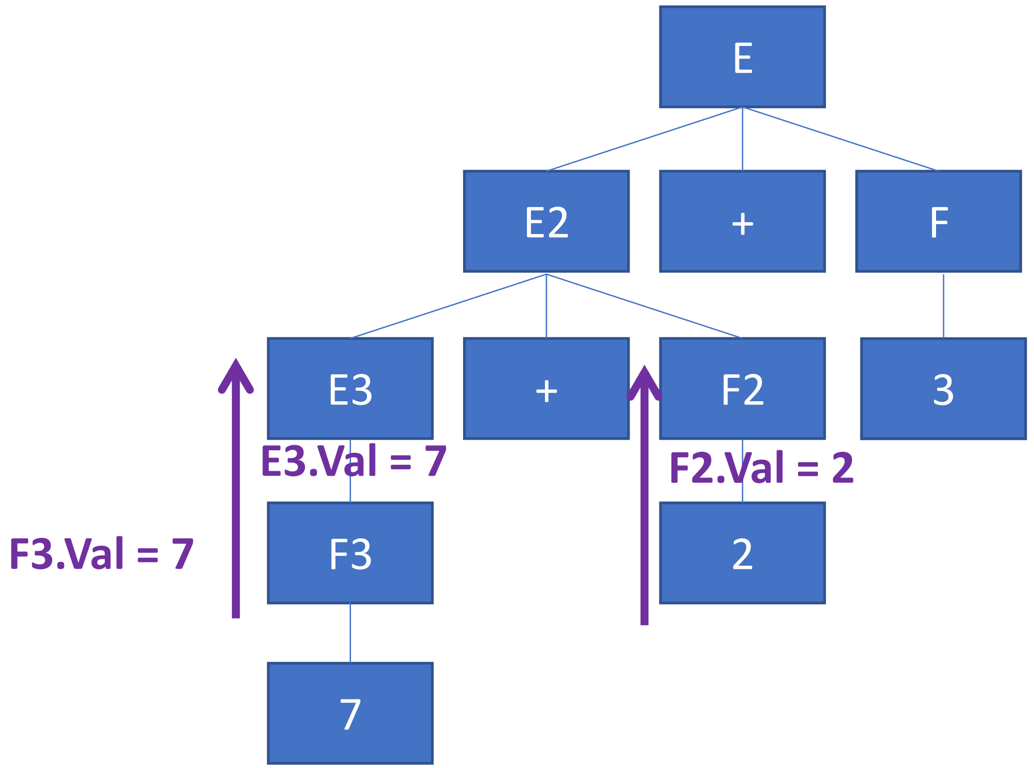
We can define a synthesized attribute 'value' for all Ex and Fx symbols.

Now…

Consider a production E2 → E3 + F2

The value of E2 would be calculated using the values of E3 and F2, as in E2.value = E3.value + F2.value.

This operation happens whenever the E → E + F CFG rule is used.



Syntax Directed Translation: definitions

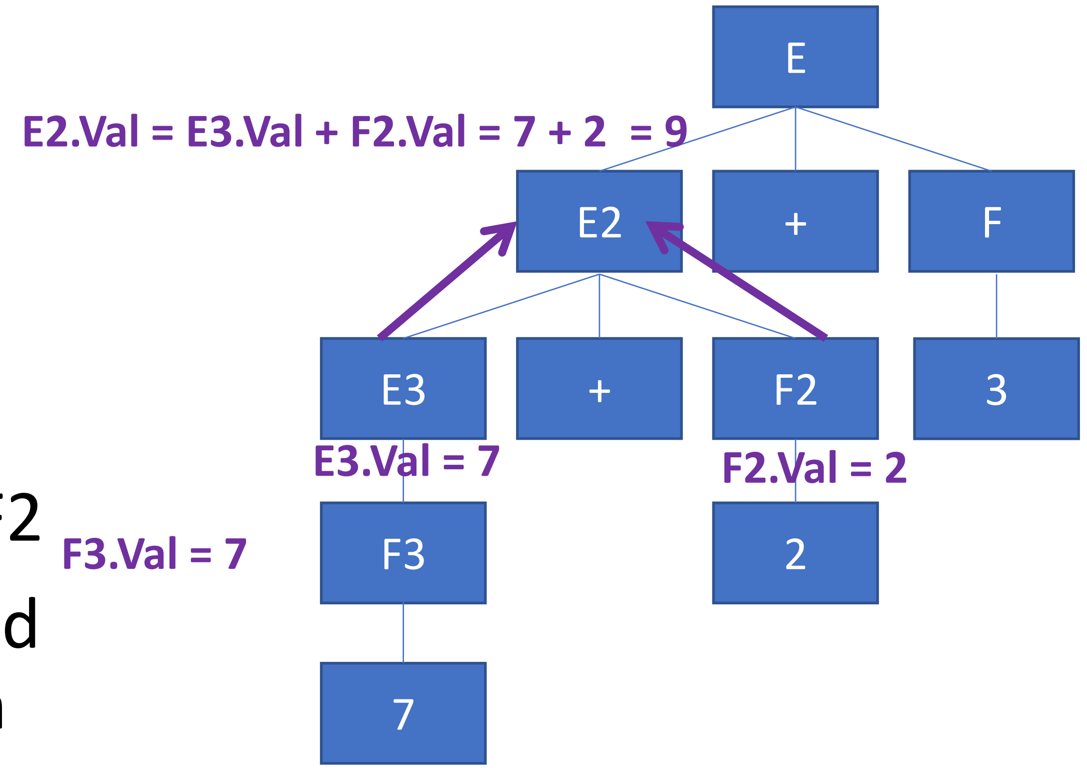
We can define a synthesized attribute 'value' for all Ex and Fx symbols.

Now…

Consider a production E2 → E3 + F2

The value of E2 would be calculated using the values of E3 and F2, as in E2.value = E3.value + F2.value.

This operation happens whenever the E → E + F CFG rule is used.



Syntax Directed Translation: definitions

Definition (Rules):

Rules in SDTs associate semantic actions and attribute computations with grammar productions rules from the CFG.

They define **how attribute values are computed and propagated in the parse tree** and how they will later contribute to the translation process.

Each rule consists of an addendum, applied to any CFG production rule and usually defines a set of attribute computation equations.

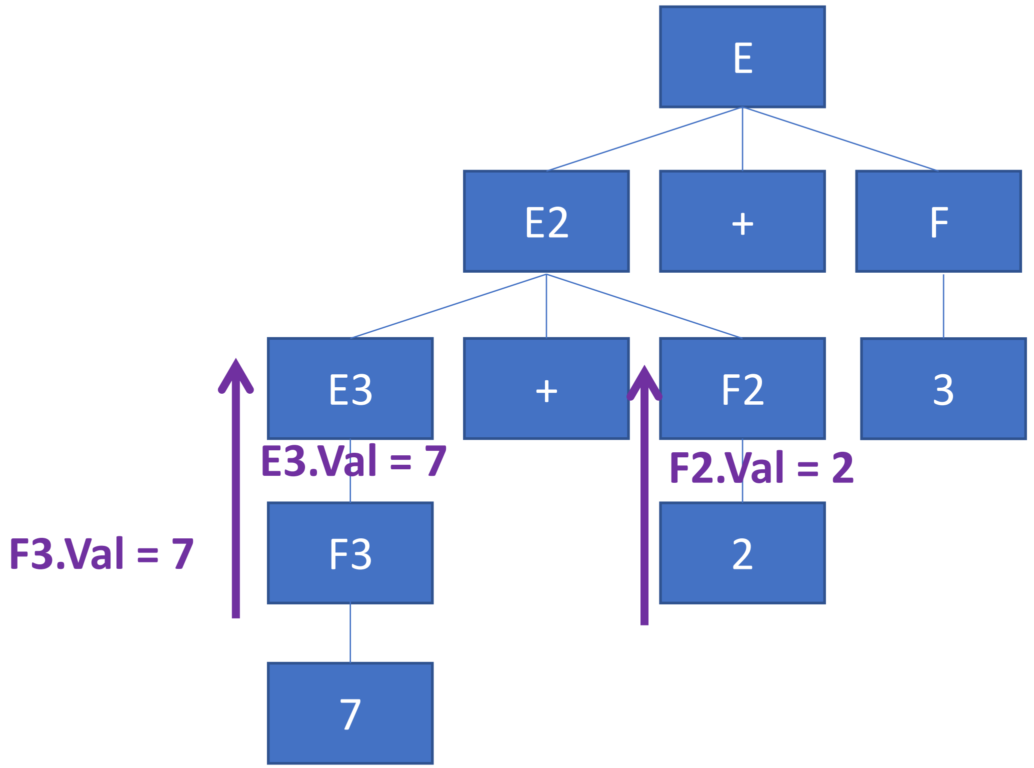
The equations specify how the attributes of the non-terminal symbols in the production are computed from the attributes of their children or from the attributes of the parent and siblings.

Syntax Directed Translation: definitions

In this previous example…

E2.value = E3.value + F2.value

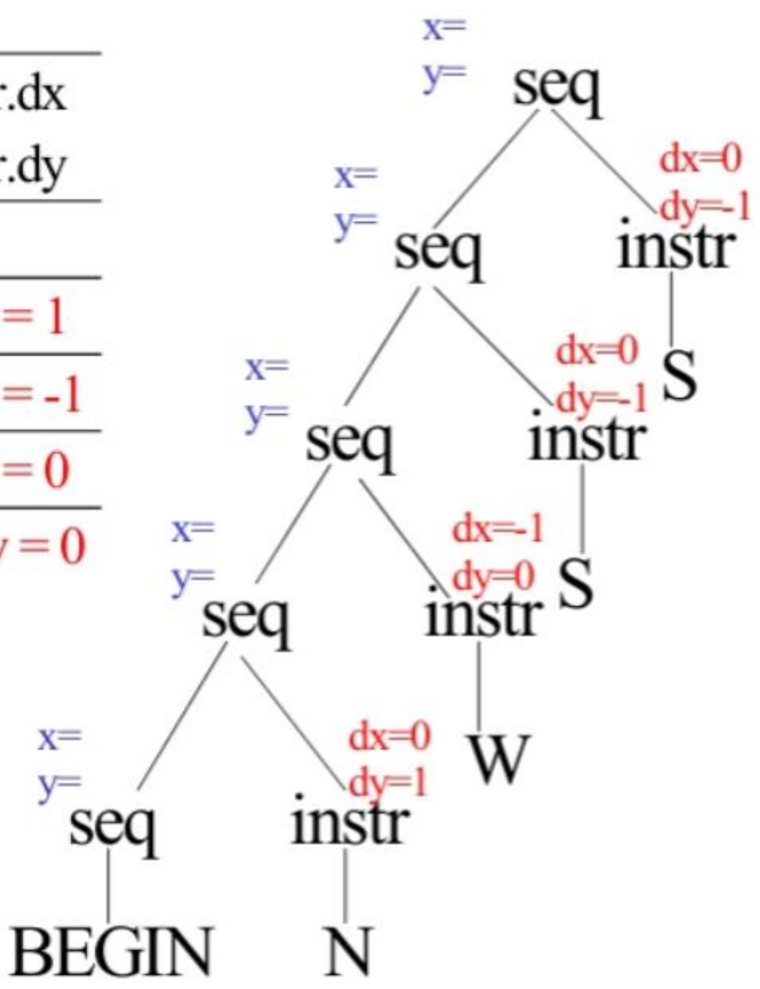
Was the SDT rule added to the production rule E → E + F.



In short, each CFG rule comes with SDT instructions to be executed in the AST!

Production	Semantic Actions
seq → seq_1 instr	seq.x = seq_1.x + instr.dx seq.y = seq_1.y + instr.dy
seq → BEGIN	seq.x = 0, seq.y = 0
instr → NORTH	instr.dx = 0, instr.dy = 1
instr → SOUTH	instr.dx = 0, instr.dy = -1
instr → EAST	instr.dx = 1, instr.dy = 0
instr → WEST	instr.dx = -1, instr.dy = 0

Input: BEGIN N W S S



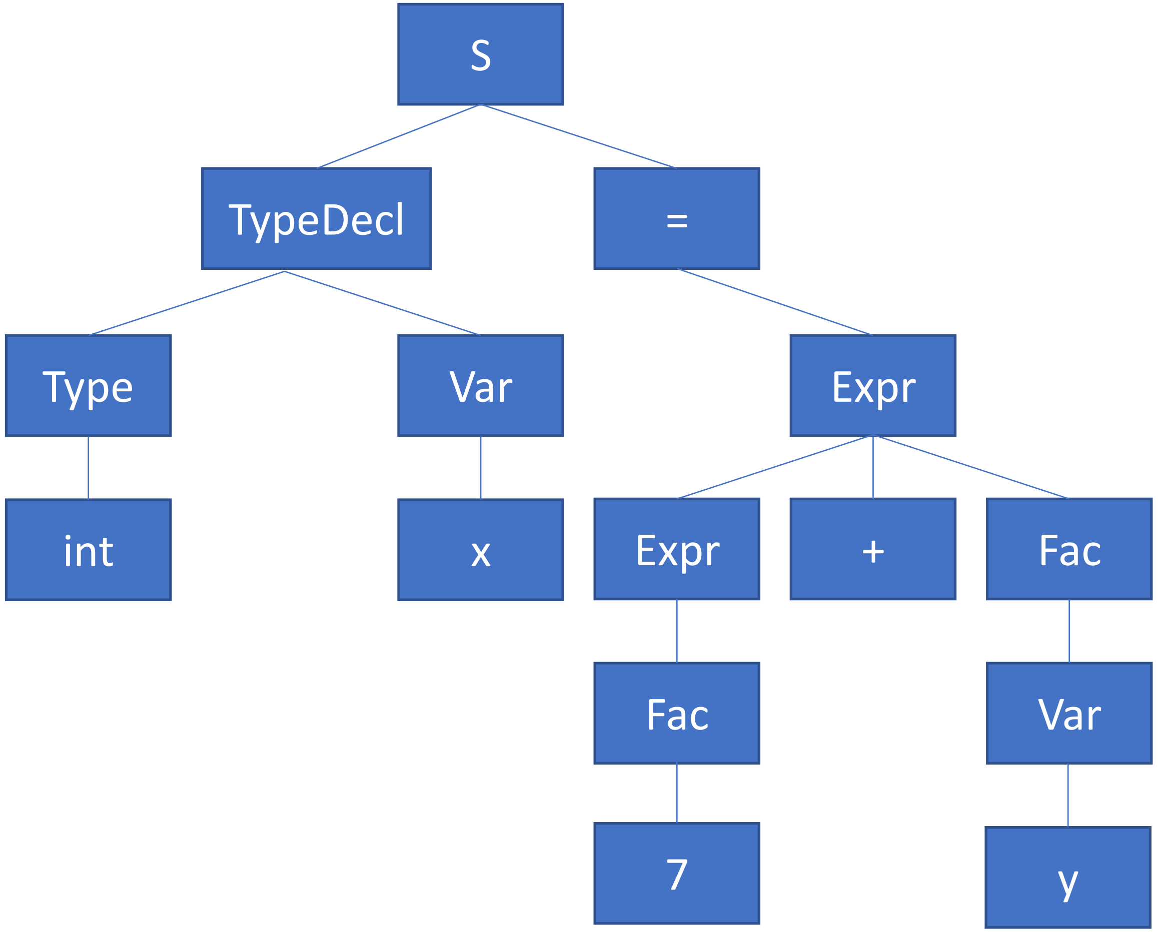
A note on inherited attributes (out of scope)

Definition (Inherited Attributes):

Inherited attributes are computed from the attribute values of the parent nodes and siblings nodes in the parse tree.

They **propagate information around and down the tree**, from the root (start symbol) towards the leaves (terminals).

Inherited attributes are useful for passing context information from higher-level constructs to lower-level constructs in the parse tree.



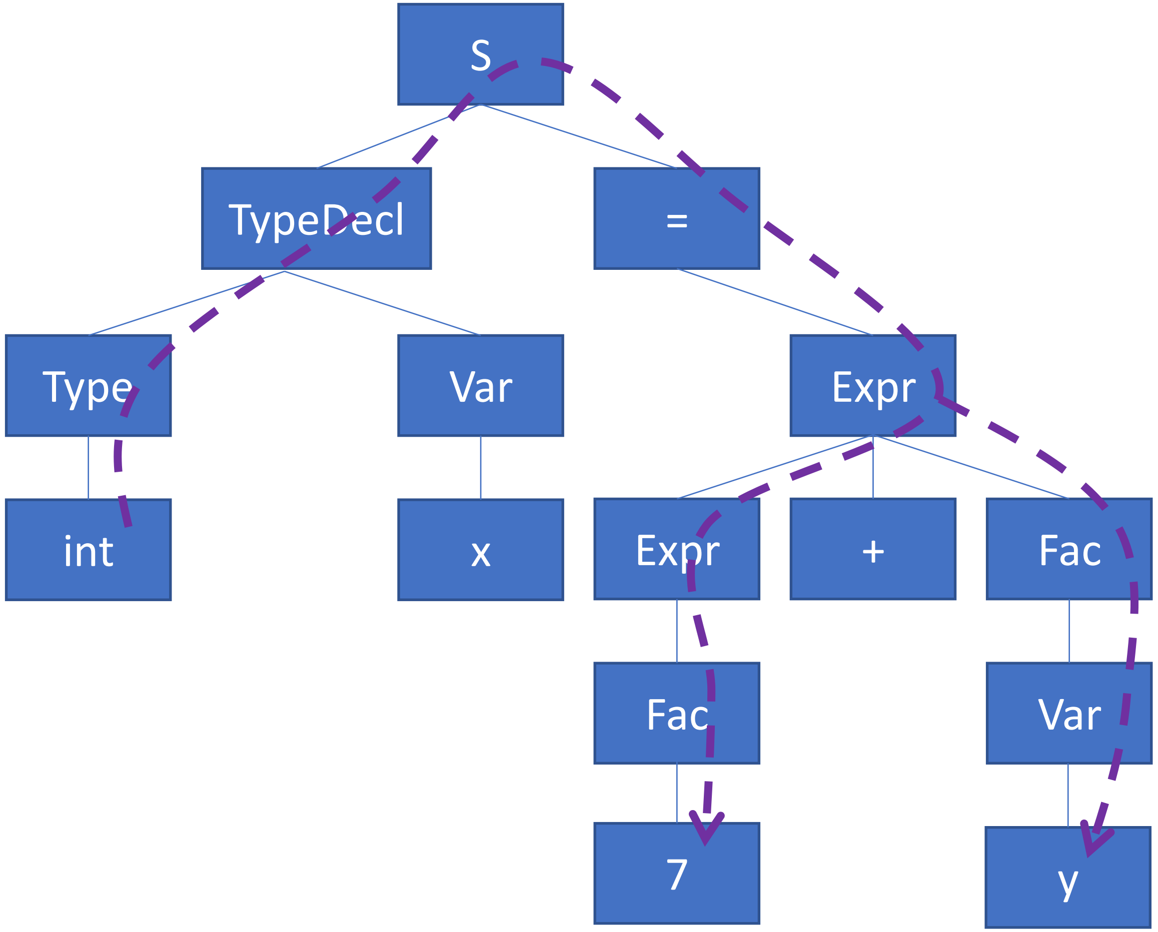
A note on inherited attributes (out of scope)

For example, consider the expression below.

int x = 7 + y;

The “int” declaration, **probably resolved last in the parse tree** for this expression, enforces constraints on the types of both the literal “7” and the identifier “*y*”, which will both need to be of type int as well.

This information will typically be propagated around the tree.



Implementing an SDT for an Arithmetic CFG

Consider the CFG below.

$$E \rightarrow E + T$$

$$E \rightarrow T$$

$$T \rightarrow \textit{(any number)}$$

We have used this one many times to check the syntax of simple arithmetic strings.

Restricted

Implementing an SDT for an Arithmetic CFG

Consider the CFG below.

It can be modified to include the following SDT rules.

$E \rightarrow E + T$	$(E.val = E.val + T.val)$
$E \rightarrow T$	$(E.val = T.val)$
$T \rightarrow$ *(any number)*	$(T.val = number.val)$

These rules specify how to compute the value of each non-terminal symbol based on the values of its children.

(Could be extended to more arithmetic operations…)

Implementing an SDT for an Arithmetic CFG

Consider the CFG below, modified with the following SDT rules.

$$E \rightarrow E + T$$
$$(E.val = E.val + T.val)$$

$$E \rightarrow T$$
$$(E.val = T.val)$$

$$T \rightarrow (\text{any number})$$
$$(T.val = number.val)$$

This CFG/SDT could later be extended to

- **More arithmetic operations**,
- **More programming operations**, (assignments, function declarations, etc.)
- **Block structures** (if, for, while, switch, etc.)
- Etc.

Implementing an SDT for an Arithmetic CFG

Basically, all the production rules of the YACC CFG will have an SDT rule implemented! (And we are quite grateful, we do not have to figure them out ourselves, even though we could!)

Production	Semantic Rules
S → **id** := E	S.code := E.code || gen(**id**.place ':=' E.place)
E → E_1 + E_2	E.place := newtemp; E.code := E_1.code || E_2.code || gen(E.place ':=' E_1.place '+' E_2.place)
E → E_1 * E_2	E.place := newtemp; E.code := E_1.code || E_2.code || gen(E.place ':=' E_1.place '*' E_2.place)

AST building procedure

To demonstrate how to build an Abstract Syntax Tree (AST), let us consider an arithmetic CFG and its SDT rules, and an arithmetic expression, for instance the string x = “26*5 + 7”. Several steps are involved:

- Use the CFG production rules to find a derivation for string x.
- If no derivation can be found, stop, because invalid syntax in x.
- Otherwise, list the production rules you have used.
- Build the parse tree for this derivation.
- Transform the list of production rules into a list of SDT rules to use on the parse tree to propagate information.
- Produce an AST, which is the propagated version of the parse tree.

Implementing an SDT for Arithmetic CFG

Consider the CFG below, modified with the following SDT rules.

E → E + T
(E.val = E.val + T.val)

E → T
(E.val = T.val)

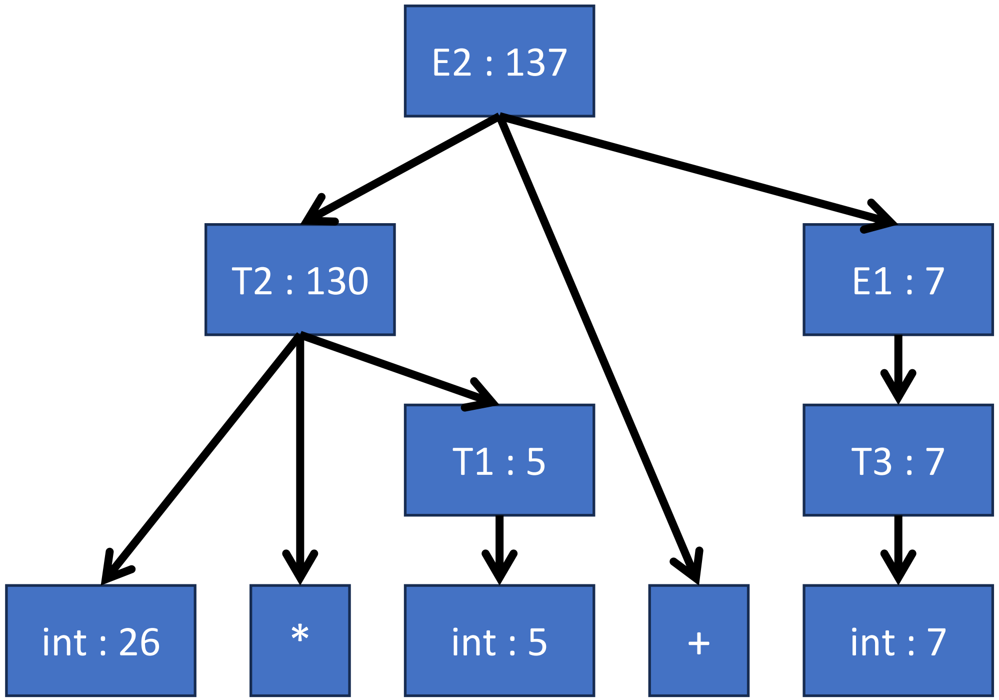
T → (any number)
(T.val = number.val)

This CFG/SDT could later be extended to

- **More arithmetic operations**,
- **More programming operations**, (assignments, function declarations, etc.)
- **Block structures** (if, for, while, switch, etc.)
- Etc.

Implementing an SDT for Arithmetic CFG

Using these steps, on the string "26*5 + 7", then gives the following Abstract Syntax Tree and list of SDT rules…



And the list of operations below:

I1.val = 26

I2.val = 5

T1.val = I2.val

*T2.val = T1.val*I1.val*

I3.val = 7

T3.val = I3.val

E1.val = T3.val

E2.val = T2.val + E1.val

From CFG to SDT

Syntax-directed definitions provide a systematic way of associating

- **Semantic actions,**
- **And attributes computations rules,**
- **With CFG productions rules.**

This SDT approach allows to build our Abstract Syntax Tree from any valid CFG derivation for the code to be checked.

This makes it possible **to define a translation process that generates target language representations from source language constructs in a well-defined manner**.

Remember: Intermediate code representations

Definition (intermediate code representations):

The intermediate code generated during this phase is often represented in a language that is easier to manipulate than the original source code.

A typical example of an intermediate code representation language that we will investigate in an upcoming lecture is the **three-address code.**

Remember: Three-address code representation

Definition (Three-address code representation):

Three-address code is a low-level intermediate code representation used by compilers to facilitate optimization and code generation.

It is called **"three-address"** because each instruction in the code **can have at most three operands**.

A typical three-address code instruction has the following format:

$$operand1 = operand2 \; operator \; operand3$$

Remember: Three-address code representation

For instance, if we consider the C code below, we would like it to be transformed...

...into its equivalent three-address code representation, below.

```c
#include <stdio.h>

int main() {
   int x = 10;
   int y = x + 5;
   printf("The value of y is %d\n", y);
   return 0;
}
```

```
t1 = 10
t2 = t1 + 5
y = t2
printf("The value of y is %d\n", y)
```

From CFG to SDT **to IR?!**

In our previous example, we obtained the following list of SDT operations and AST, shown on the right…

These SDT operations, are technically something we could **easily translate into a three-address representation**, following the syntax below!

$$operand1 = operand2\ op\ operand3$$

- I1.val = 26
- I2.val = 5
- T1.val = I2.val
- T2.val = T1.val*I1.val
- I3.val = 7
- T3.val = I3.val
- E1.val = T3.val
- E.val = T2.val + E1.val

From CFG to SDT **to IR?!**

Having an **Abstract Syntax Tree**, following from using SDT Rules on our Parse Tree/CFG, **brings us very close to being able to produce an Intermediate Code Representation!**

- I1.val = 26
- I2.val = 5
- T1.val = I2.val
- T2.val = T1.val*I1.val
- I3.val = 7
- T3.val = I3.val
- E1.val = T3.val
- E.val = T2.val + E1.val

→

- i1 = 26
- i2 = 5
- t1 = i2
- t2 = t1*i1
- i3 = 7
- t3 = i3
- e1 = t3
- e = t2 + e1

From CFG to SDT **to IR?!**

Having an **Abstract Syntax Tree**, following from using SDT Rules on our Parse Tree/CFG, **brings us very close to being able to produce an Intermediate Code Representation!**

Two problems, however…

- I1.val = 26
- I2.val = 5
- T1.val = I2.val
- T2.val = T1.val*I1.val
- I3.val = 7
- T3.val = I3.val
- E1.val = T3.val
- E.val = T2.val + E1.val

- i1 = 26
- i2 = 5
- t1 = i2
- t2 = t1*i1
- i3 = 7
- t3 = i3
- e1 = t3
- e = t2 + e1

Problem #1

Problem #1:
Valid Lexemes + Valid Syntax
$\neq$ Valid Code.

```
#include <stdio.h>
#include <stdlib.h>

int y = x + 7;
int x = 2;
```

For instance, the code snippet shown on the right has:

- Valid lexemes,
- A valid syntax.

But it is not a valid code to execute (why?)

Problem #1

Problem #1:
Valid Lexemes + Valid Syntax
$\neq$ Valid Code.

A code with valid lexemes and a valid syntax **could still be illegal for semantic reasons**, e.g. variables undeclared and called.

Need an extra semantic analysis before claiming the code is legal and can be transformed into IR!

```c
#include <stdio.h>
#include <stdlib.h>

int y = x + 7;
int x = 2;
```

Windows PowerShell

```
PS D:\Shortcuts\Desktop> gcc main.c -o output_file_name
main.c:5:9: error: 'x' undeclared here (not in a function)
 int y = x + 7;
         ^
PS D:\Shortcuts\Desktop> _
```

Problem #2

Given a CFG and a string *x* whose syntax needs to be verified

- We understand how to use CFG production rules and apply them manually to eventually find the string *x*.
- We understand how to derive a parse tree for the derivation.
- We understand how to add syntax-directed rules to our production rules and use them in the derivation.
- We understand how to use syntax-directed rules to transform our parse tree into an AST.

It is probably not that hard to automate the process that will give us the valid CFG derivation for any input string and its AST, right?

Problem

Given a CFG d

- We unders y them manually t
- We unders on.
- We unders roduction rules and u
- We unders orm our parse tree

It is probabl vill give us the valid CF ght?

Problem #2

It is probably not that hard to automate the process that will give us the valid CFG derivation for any input string and its AST, right?

Oh my dear, if only…

Problem #2

For syntax analysis, we still need

- **An algorithm to find if a given string of tokens *x* has a valid syntax and admits a valid derivation** for our given CFG,
- And one that will **return the derivation, parse tree and abstract syntax tree** produced for any syntactically valid string *x*!

Small tiny teeny issue: This is absolutely not an easy problem algorithmically speaking! (It is very much NP-Hard!)
Will take us the next three lectures…!
And we will not even dare to implement said algorithms…!

Quiz time!

Which problem can be caused by ambiguous grammars in compilers?

A. Performance issues

B. Parsing errors and intermediate code being incorrectly computed

C. A valid source code being recognized as faulty by the compiler

D. None of the above

Quiz time!

Which problem can be caused by ambiguous grammars in compilers?

A. Performance issues

B. Parsing errors and intermetidate code being incorrectly computed

C. A valid source code being recognized as faulty by the compiler

D. None of the above

Quiz time!

What is the main goal of Syntax-Directed Translation (SDT)?

A. To create an efficient parser

B. To translate operations in a source code, in a sequence of simple operations, that can later easily translate into intermediate code

C. To optimize the source code

D. To detect syntax errors in the source program

Quiz time!

What is the main goal of Syntax-Directed Translation (SDT)?

A. To create an efficient parser

B. To translate operations in a source code, in a sequence of simple operations, that can later easily translate into intermediate code

C. To optimize the source code

D. To detect syntax errors in the source program

Quiz time!

In Syntax-Directed Translation, what are inherited attributes?

A. Attributes that are passed from parent to child nodes
B. Attributes that are passed from child to parent nodes
C. Attributes that are shared between sibling nodes
D. Attributes that are only associated with terminal nodes

Quiz time!

In Syntax-Directed Translation, what are inherited attributes?

A. **Attributes that are passed from parent to child nodes**
B. Attributes that are passed from child to parent nodes
C. Attributes that are shared between sibling nodes
D. Attributes that are only associated with terminal nodes

Quiz time!

What is a great benefit, following from using Abstract Syntax Trees compared to parse trees only?

A. ASTs include only the essential information required to compute a given expression and break the computation into elementary steps
B. ASTs are guaranteed to be symmetrical, while parse trees are not
C. ASTs and parse trees are the same
D. Both A and B

Quiz time!

What is a great benefit, following from using Abstract Syntax Trees compared to parse trees only?

A. **ASTs include only the essential information required to compute a given expression and break the computation into elementary steps**
B. ASTs are guaranteed to be symmetrical, while parse trees are not
C. ASTs and parse trees are the same
D. Both A and B

Quiz time!

In programming languages, which of the following can be represented using an Abstract Syntax Tree?

A. Arithmetic expressions

B. Boolean expressions

C. Simple programming language constructs, like if and while

D. All of the above

E. None of the above

Quiz time!

In programming languages, which of the following can be represented using an Abstract Syntax Tree?

A. Arithmetic expressions

B. Boolean expressions

C. Simple programming language constructs, like if and while

D. All of the above

E. None of the above

If time allows, practice

Practice 1: Precedence and associativity

Consider the three Boolean expressions below.

A. TRUE AND FALSE OR NOT TRUE
B. NOT (TRUE OR FALSE) AND TRUE
C. (NOT TRUE OR FALSE) AND (TRUE OR NOT FALSE)

Question: What is the precedence order between the different operators in these expressions? For each of the typical Boolean operations, are they left-associative or right-associative?

If time allows, practice

Practice 2: Grammar Rewriting

Consider the CFG below, for Boolean expressions, which will check any string of Booleans consisting of combinations of the words true, false, and, or and not, along with parentheses. As or, is now a keyword, we use | as in RegEx to separate different possible outputs for prod. rules.

E → E and E | E or E | not E | (E) | T

T → true | false

Question: Is this CFG ambiguous?

If so, can you provide an example of a string that generates two parse trees with opposite behaviors?

And if so, can you rewrite the CFG in a non-ambiguous manner?

If time allows, practice

Practice 2: Grammar Rewriting

Answer: Yes it is ambiguous. The problem has to do with precedence between and/or/not and many problematic strings can be considered.

A possible way to rewrite the CFG to make it not ambiguous is shown below. It accounts for associativity and precedence on each operator.

E → E or T | T

T → T and F | F

F → not F | (E) | V

V → true | false

If time allows, practice

Practice 3: From CFG to SDT

Question: Consider the CFG below, being the non-ambiguous answer provided to the previous question. Which SDT rules would you add to each of the production rules below?

E → E or T

E → T

T → T and F

T → F

F → not F

F → (E)

F → L

L → true | false

If time allows, practice

Practice 4: SDT, and AST construction

Consider the CFG and SDT rules (between {}) below.

$$E \rightarrow E + T \ \{E.val = E1.val + T.val\}$$

$$E \rightarrow T \ \{E.val = T.val\}$$

$$T \rightarrow T * F \ \{T.val = T1.val * F.val\}$$

$$T \rightarrow F \ \{T.val = F.val\}$$

$$F \rightarrow (E) \ \{F.val = E.val\}$$

$$F \rightarrow num \ \{F.val = num.val\}$$

Question: Build an AST and find the list of three-address operations resulting from the arithmetic string “(4+3)*2 + 8*7 + 3”.