Compilers: Symbol Collection and Type Checking a topic in

DM565 – Formal Languages and Data Processing

Kim Skak Larsen

Department of Mathematics and Computer Science (IMADA)
University of Southern Denmark (SDU)

kslarsen@imada.sdu.dk

October, 2023

Compiler Phases (Recap)

Scanner

Parser

Symbol Collection Type Checking

Code Generation Emit

Scanner (lexical analysis)

We have shown that

regular expressions can be used to specify tokens, everything can be transformed to and combined into finite automata, and

using appropriate ways to run the resulting DFA, we can find and return tokens.

We have seen that we can use tools to do this for us: flex (lex) for C (or Java, etc.) and

the ply package for Python.

Compiler Phases (Recap) SDU

Scanner

Parser

Symbol Collection Type Checking

Code Generation Emit

Parser (syntax analysis)

We have shown that we can

construct top-down (predictive, LL(1)) parsers via the concepts of

NULLABLE, FIRST, and FOLLOW, and the construction of either

a predictive parsing table, or
a recursive descent program.

construct bottom-up (LR(1)/LALR(1)) parsers by computing LR(1) states, and then

transforming into LR(1)/LALR(1) parsing tables.

We have seen that for bottom-up parsing, we can use tools to do this for us:

bison (yacc) for C (or Java, etc.) and

the ply package for Python.

Parser

Symbol Collection

Type Checking

Code Generation

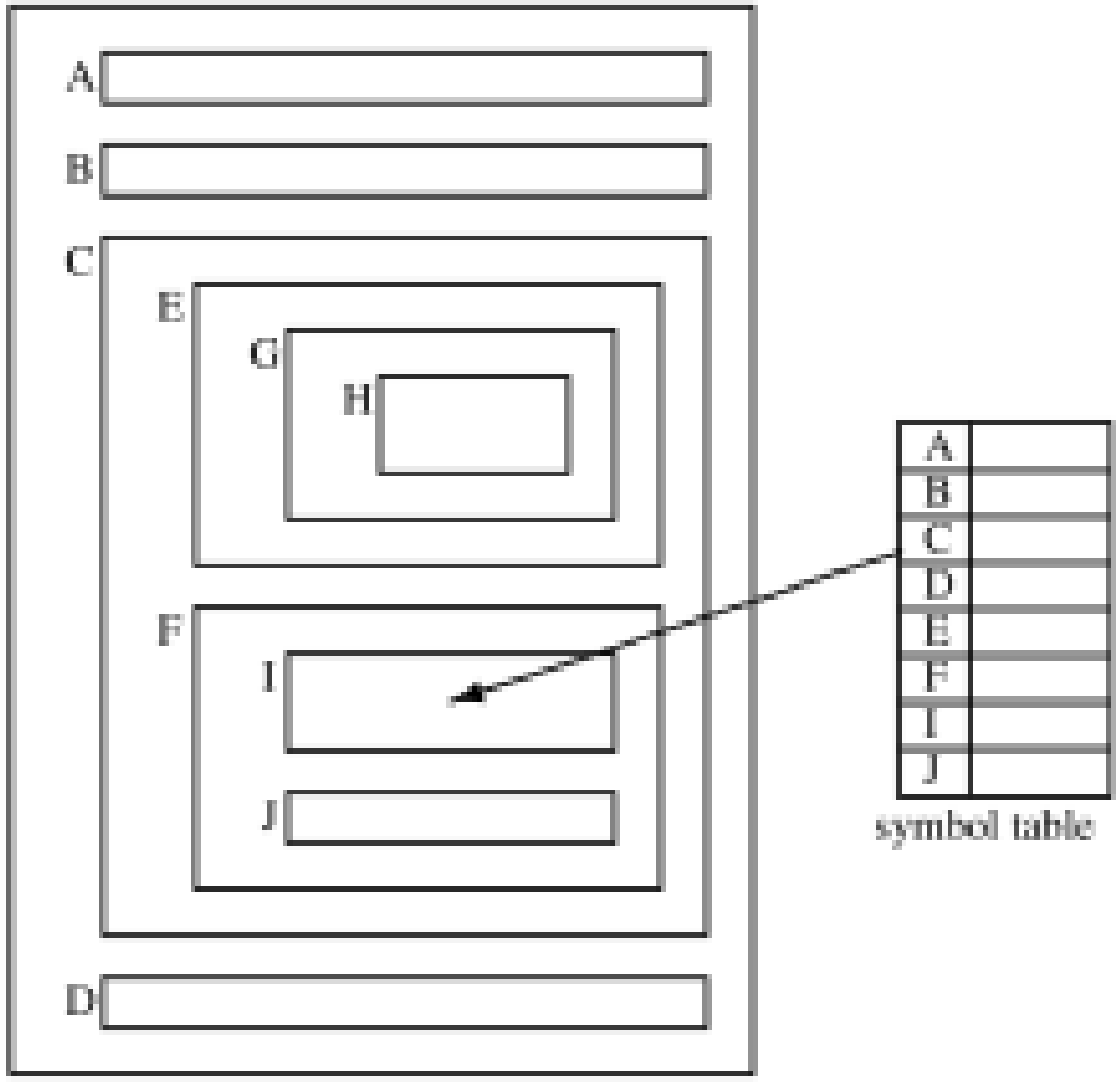
Emit

Scope: Scil example SDU

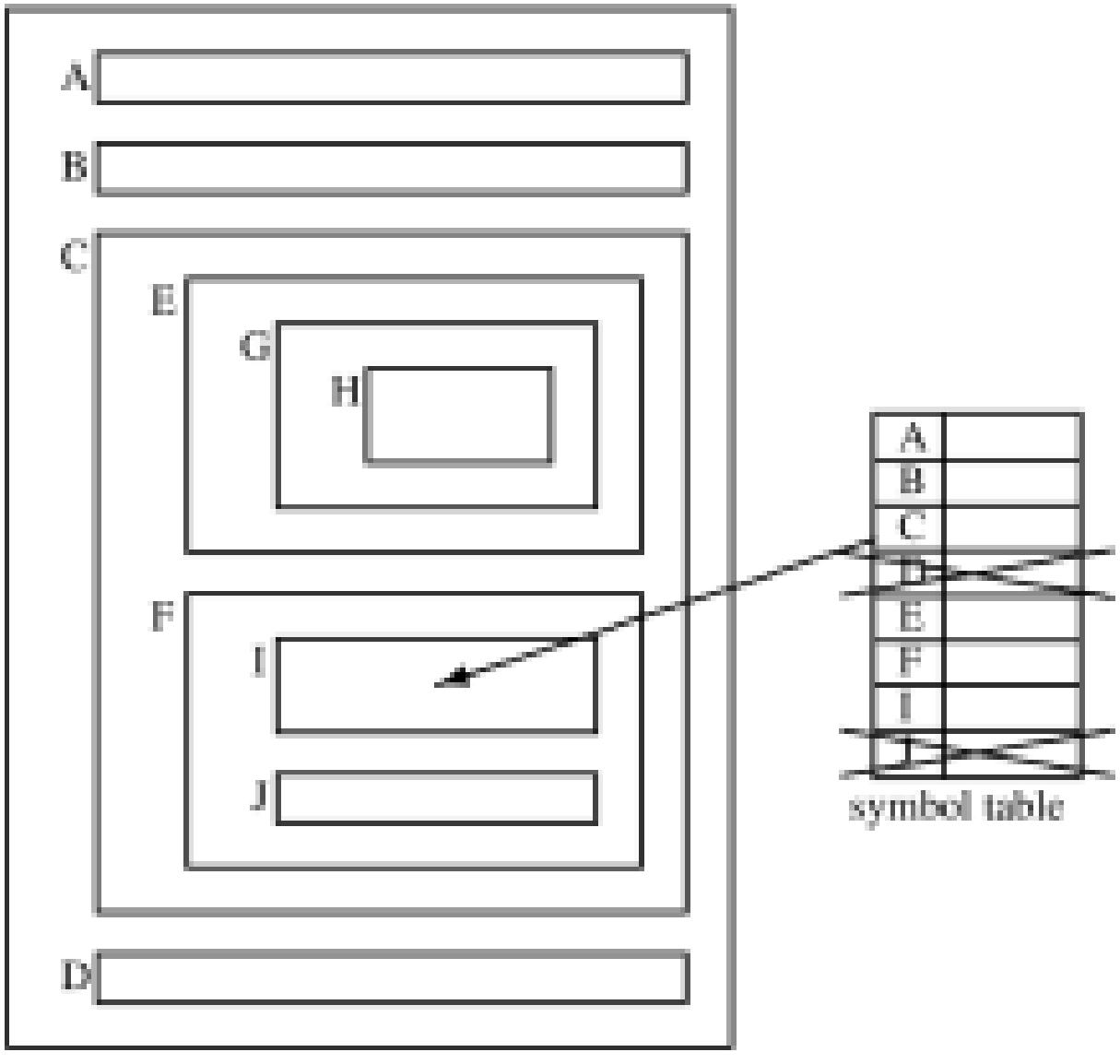
```
var a , dummy
```

```
function f (){

    function g (){

        var a

        function h (){
            a = 0;
            print a ; # 3 rd output : 0
        }

        a = 2; print a ; # 2 nd output : 2
        dummy = h (); print a ; # 4 th output : 0
        return 0;
    }

    print a ; # 1 st output : 1
    dummy = g (); print a ; # 5 th output : 1
    return 0;
}

a = 1; dummy = f (); print a ; # 6 th output : 1
return 0;
```

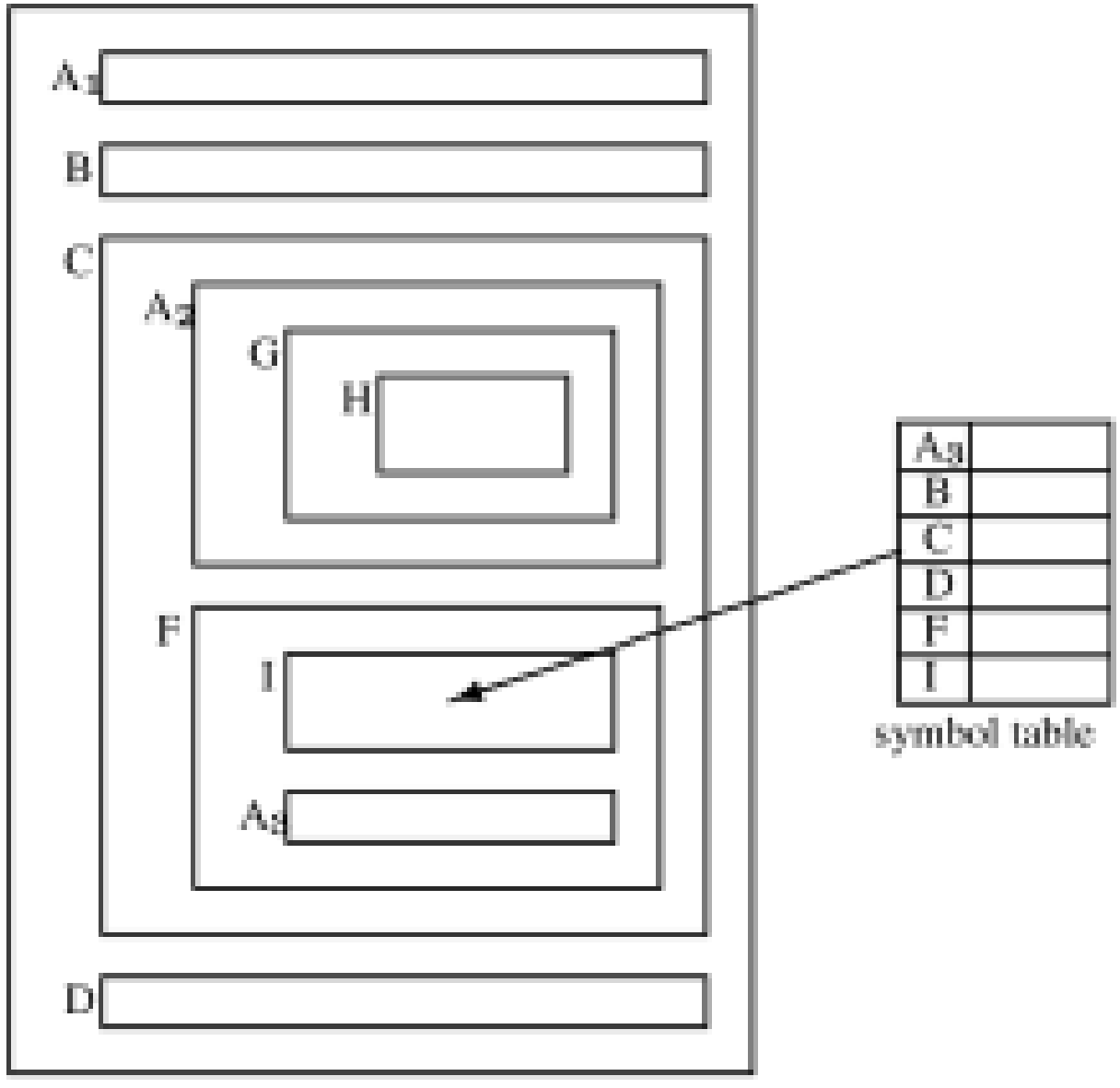

Scope: Static, nested scope SDU


A
B
C
E
G
H
F
I
J
D
A
B
C
D
E
F
I
J
symbol table

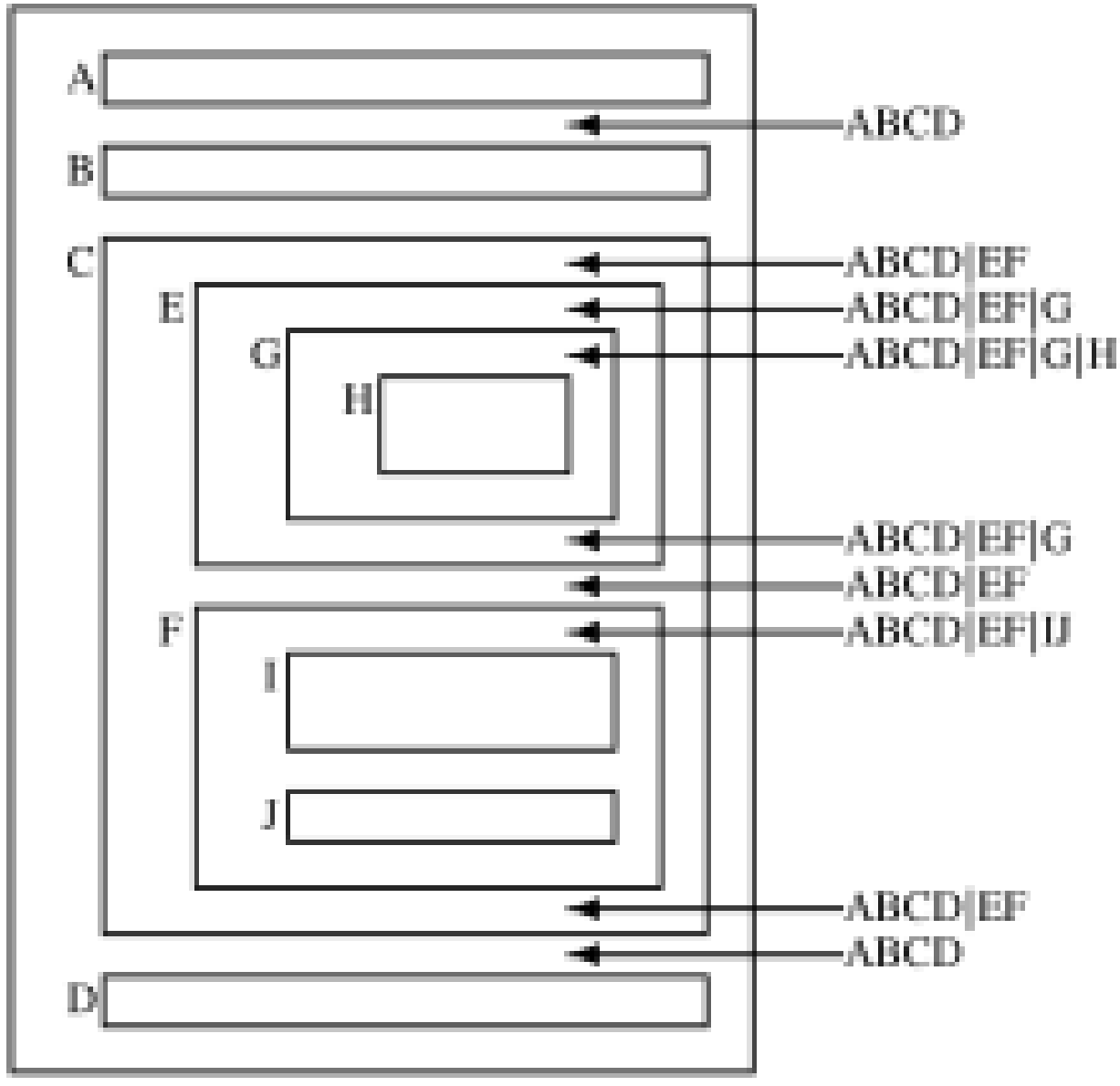
Scope: Old-fashioned SDU


A
B
C
E
G
H
F
I
J
D
A
B
C
D
E
F
I
symbol table

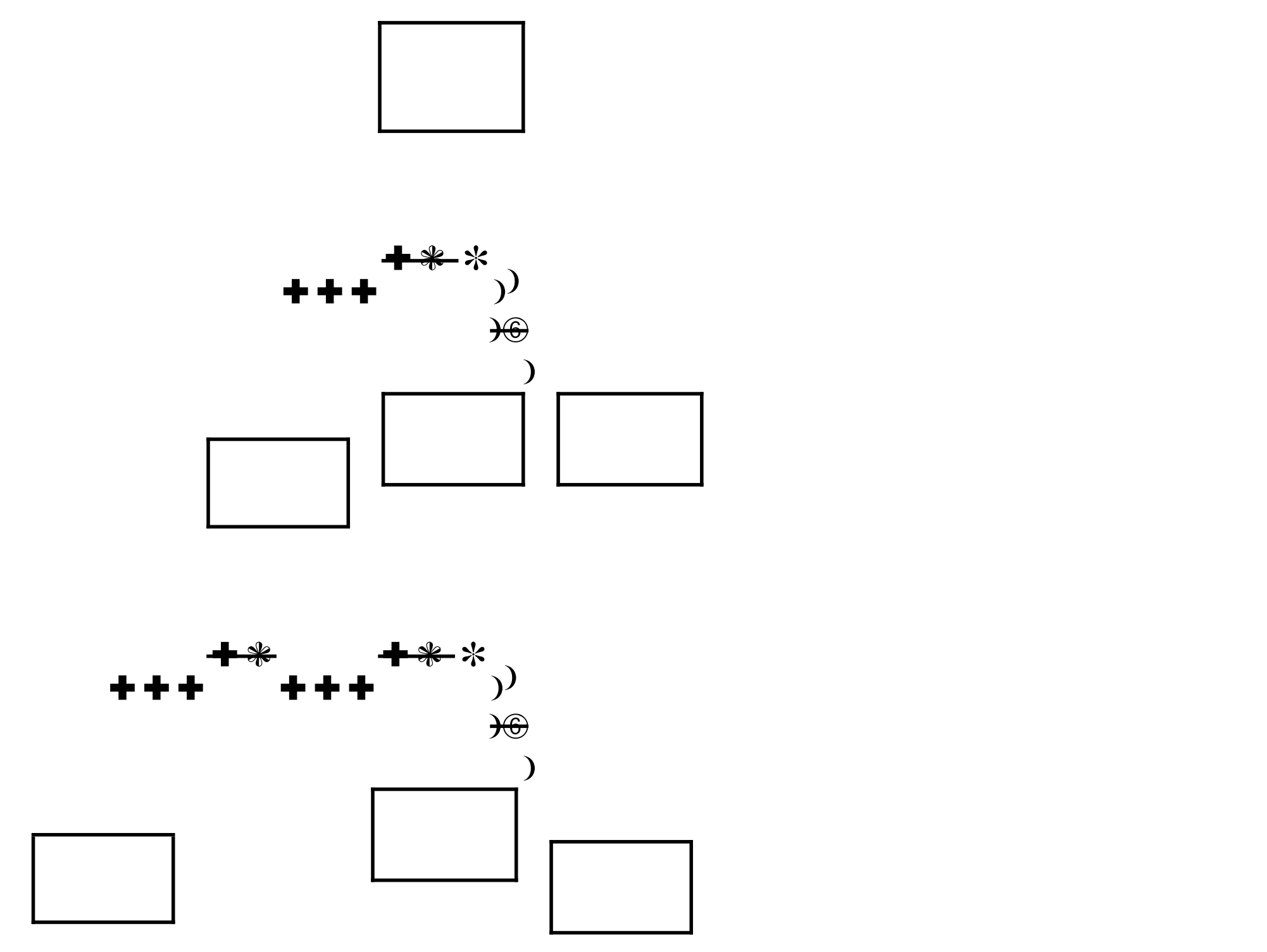
Scope: Name hiding SDU

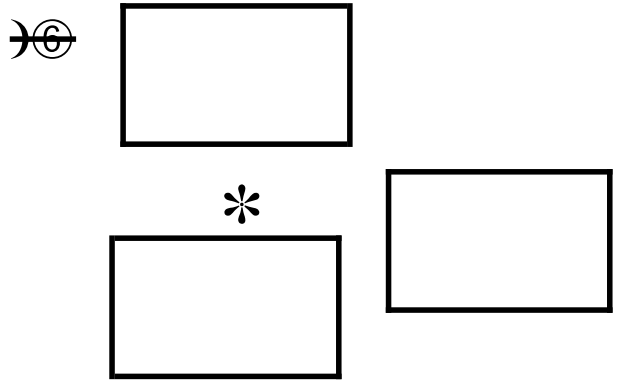

A1
B
C
A2
G
H
F
I
A3
D
A3
B
C
D
F
I
symbol table

Scope: Stack behavior



Symbol Table Structure





Symbol Table Realization in C (example)

```
# define HashSize 317
# define NEW ( type ) ( type *) malloc ( sizeof ( type ))
void * malloc ( unsigned n );

typedef struct SYMBOL {
    char * name ;
    int value ;
    struct SYMBOL * next ;
```

```
} SYMBOL ;

typedef struct SymbolTable {
        SYMBOL * table [ HashSize ];
        struct SymbolTable * parent ;
} SymbolTable ;

int Hash ( char * str );

SymbolTable * initSymbolTable ( SymbolTable * t );

SYMBOL * insert ( SymbolTable *t , char * name , int value );

SYMBOL * lookup ( SymbolTable *t , char * name );
```


Symbol Table Realization in Python (example)

SDU

```
class SymbolTable :
        " " " Implements a classic symbol table for static nested
              scope . Names for each scope are collected in a
              Python dictionary . The parent scope can be accessed
              via the parent reference .
        " " "
        def __init__ ( self , parent ):
```

```
        self . _tab = {}
        self . parent = parent

def insert ( self , name , value ):
        self . _tab [ name ] = value

def lookup ( self , name ):
        if name in self . _tab :
                return self . _tab [ name ]
        elif self . parent :
                return self . parent . lookup ( name )
        else :
                return None
```


Compiler Phases

Scanner

Parser

Symbol Collection

Type Checking

Code Generation

Emit

Type Checking

In addition to simple type rules covered in the lecture notes, advanced type checking questions such as those related to structural equivalence can be answered using our DFA tool base.