TDDD55- Compilers and Interpreters
Lesson 3

Zeinab Ganjei (zeinab.ganjei@liu.se)

Department of Computer and Information Science
Linköping University

1. Grammars and Top-Down Parsing

- Some grammar rules are given
- Your task:
  - Rewrite the grammar (eliminate left recursion, etc.)
  - Add attributes and attribute rules to the grammar
  - Implement your grammar in a C++ class named **Parser**. The **Parser** class should contain a method named **Parse** that returns the value of a single statement in the language.

2. Scanner Specification

- Finish a scanner specification given in a *scanner.l* flex file, by adding rules for C and C++ style comments, identifiers, integers, and floating point numbers.

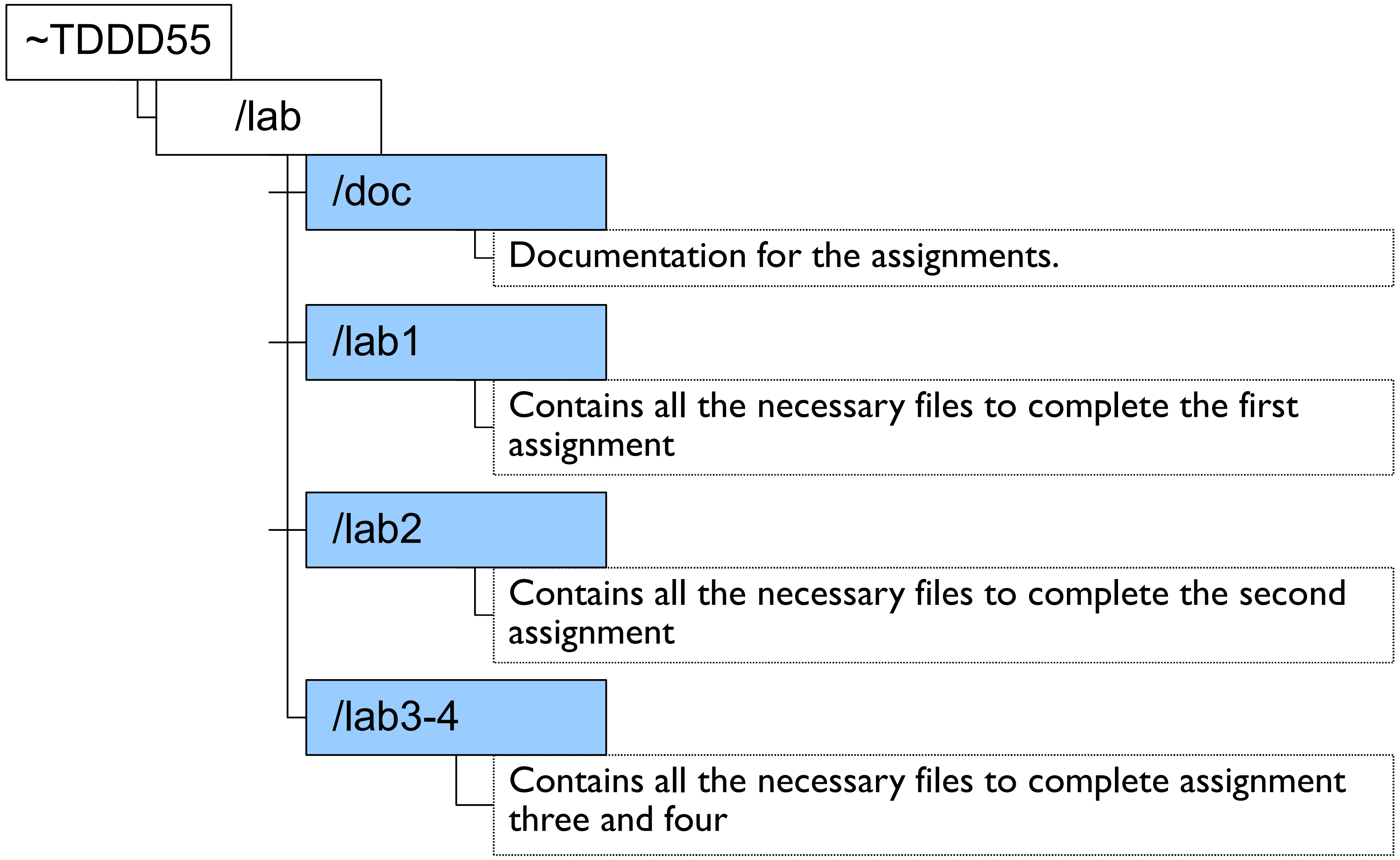
3. Parser Generators

- Finish a parser specification given in a *parser.y* bison file, by adding rules for expressions, conditions and function definitions, ... You also need to augment the grammar with error productions.

4. Intermediate Code Generation

- The purpose of this assignment is to learn about how abstract syntax trees can be translated into intermediate code.
- You are to finish a generator for intermediate code by adding rules for some language statements.

Laboratory Skeleton


~TDDD55
/lab
/doc
Documentation for the assignments.
/lab1
Contains all the necessary files to complete the first assignment
/lab2
Contains all the necessary files to complete the second assignment
/lab3-4
Contains all the necessary files to complete assignment three and four

Bison – Parser Generator

Purpose of a Parser

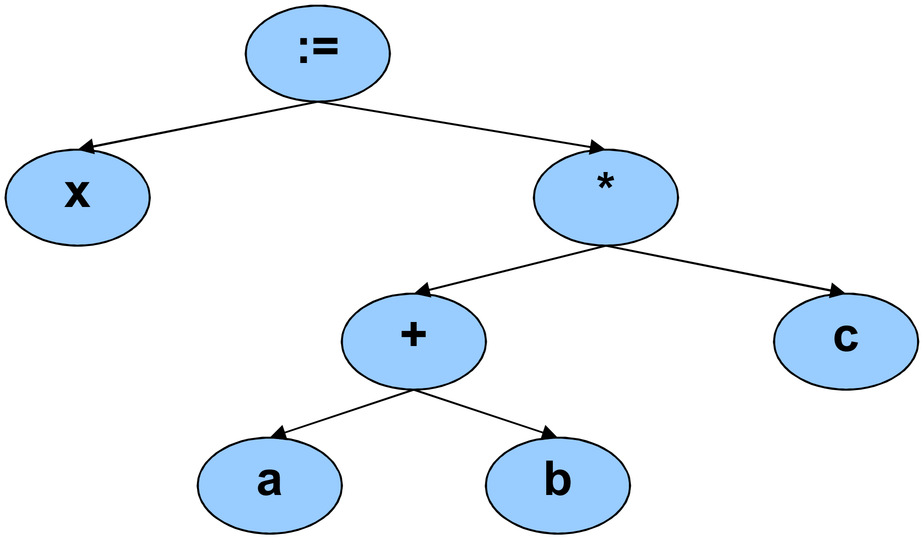
- The parser accepts tokens from the scanner and verifies the syntactic correctness of the program.
  - Syntactic correctness is judged by verification against a formal grammar which specifies the language to be recognized.
- Along the way, it also derives information about the program and builds a fundamental data structure known as parse tree or abstract syntax tree (ast).
- The abstract syntax tree is an internal representation of the program and augments the symbol table.

Bottom-Up Parsing

- Recognize the components of a program and then combine them to form more complex constructs until a whole program is recognized.
- The parse tree is then built from the bottom and up, hence the name.

Bottom-Up Parsing(2)

```
x := ( a + b ) * c;
```



LR Parsing

- A Specific bottom-up technique
  - LR stands for Left to right scan, reversed Rightmost derivation.
  - Probably the most common & popular parsing technique.
  - yacc, bison, and many other parser generation tools utilize LR parsing.
  - Great for machines, not so great for humans …

Pros and Cons of LR parsing

- Advantages of LR:
    - Accept a wide range of grammars/languages
    - Well suited for automatic parser generation
    - Very fast
    - Generally easy to maintain

- Disadvantages of LR:
    - Error handling can be tricky
    - Difficult to use manually

Bison

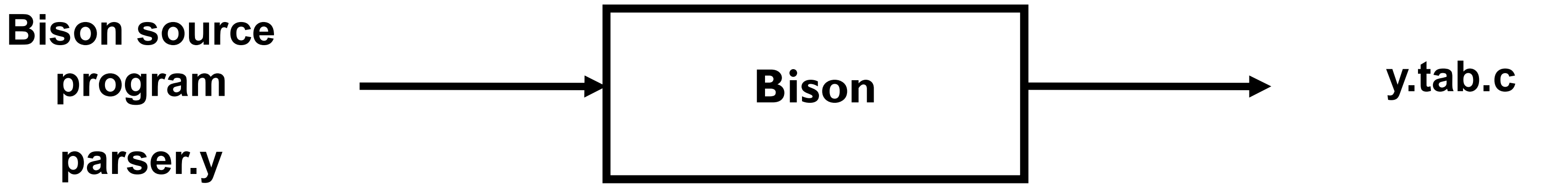
- **Bison** is a general-purpose parser generator that converts a grammar description of a context-free grammar into a **C** program to parse that grammar

Bison (2)

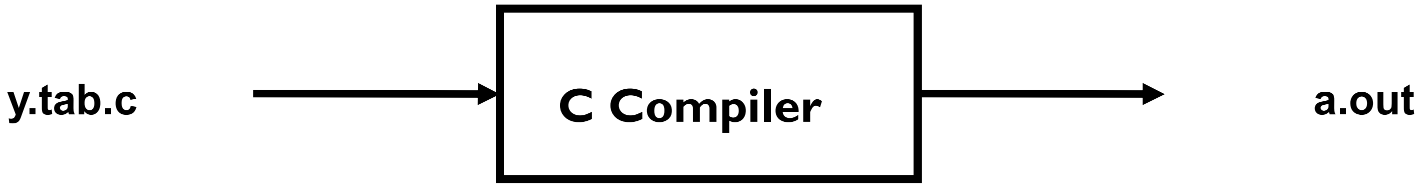
- Input: a specification file containing mainly the grammar definition
- Output: a C source file containing the parser
- The entry point is the function int yyparse();
  - yyparse reads tokens by calling yylex and parses until
    - end of file to be parsed, or
    - unrecoverable syntax error occurs
  - returns 0 for success and 1 for failure

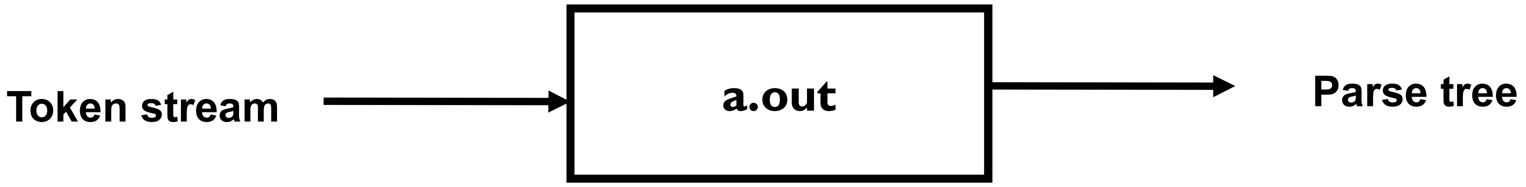
Bison Usage

- Run Bison on the grammar to produce the parser.



- Compile the code output by Bison, as well as any other source files.





Bison Specification File

- A Bison specification is composed of 4 parts.

```
%{
    /* C declarations */
%}
    /* Bison declarations */

%%

    /* Grammar rules */

%%

    /* Additional C code */
```

1.1. C Declarations

- Contains macro definitions and declarations of functions and variables that are used in the actions in the grammar rules
- Copied to the beginning of the parser file so that they precede the definition of yyparse
- Use #include<…> to get the declarations from a header file. If C declarations aren’t needed, then the %{ and %} delimiters that bracket this section can be omitted

1.2. Bison Declarations

- Contains :
  - declarations that define terminal and non-terminal symbols
  - Data types of semantic values of various symbols
  - specify precedence

Bison Specification File

- A Bison specification is composed of 4 parts.

```
%{
        /* C declarations */
%}
        /* Bison declarations */

%%

        /* Grammar rules */

%%

        /* Additional C code */
```

2. Grammar Rules

- Contains one or more Bison grammar rules, and nothing else.
- Example:
  - expression : expression ‘+’ expression { $$ = $1 + $3; } ;

- There must always be at least one grammar rule, and the first %% (which precedes the grammar rules) may never be omitted even if it is the first thing in the file.

Bison Specification File

- A Bison specification is composed of 4 parts.

```
%{
    /* C declarations */
%}
    /* Bison declarations */

%%

    /* Grammar rules */

%%

    /* Additional C code */
```

3. Additional C Code

- Copied verbatim to the end of the parser file, just as the C declarations section is copied to the beginning.
- This is the most convenient place to put anything that should be in the parser file but isn't needed before the definition of yyparse.
- For example, the definitions of yylex and yyerror often go hear

Bison Example 1 (1/2)

A parser to parse simple mathematical expressions and compute the corresponding value.

```
%{
#include <ctype.h> /* standard C declarations here */
 //extern int yylex();
}%
%token DIGIT /* bison declarations */
%%
/* Grammar rules */
line : expr '\n'       {  printf { "%d\n", $1 }; }    ;
expr : expr '+' term    {  $$ = $1 + $3;  }
       | term                                   ;
term : term '*' factor      {  $$ = $1 * $3;  }
       | factor                                 ;
```

Bison Example 1 (2/2)

```
factor : ‘(‘ expr ’)’     {  $$ = $2;  }
              | DIGIT ;
%%
/* Additional C code */

int yylex () {
   /* A really simple lexical analyzer */
   int c;
   c = getchar ();
   if ( isdigit (c) ) {
      yylval = c - ’0’ ;
      return DIGIT;
   }
   return c;
}
```

Bison Example 2 – Mid-Rules

```
thing: A { printf("seen an A"); } B ;

The same as:

thing: A fakename B ;
fakename: /* empty pattern*/ { printf("seen an A"); } ;
```

Bison Example 3 (1/2)

```
/* Infix notation calculator--calc */

%{
#define YYSTYPE double
#include <math.h>
%}

/* BISON Declarations – specifying associativity and precedence*/
%token NUM
%left '-' '+‘              /*left associative */
%left '*' '/‘
%right '^'    /* exponentiation – right associative       */

/* Grammar follows */
%%
```

Bison Example 3 (2/2)

```
input:    /* empty string */
        | input line
;
line:     '\n'
        | exp '\n'  { printf ("\t%.10g\n", $1); };
exp:      NUM                { $$ = $1;         }
        | exp '+' exp        { $$ = $1 + $3;    }
        | exp '-' exp        { $$ = $1 - $3;    }
        | exp '*' exp        { $$ = $1 * $3;    }
        | exp '/' exp        { $$ = $1 / $3;    }
        | exp '^' exp        { $$ = pow ($1, $3); }
        | '(' exp ')'        { $$ = $2;         }
;
%%
```

Syntax Errors

- Error productions can be added to the specification
- They help the compiler to recover from syntax errors and to continue to parse
- In order for the error productions to work we need at least one valid token after the error symbol, e.g. semicolon, parenthesis, etc
- Example:
    - functionCall : ID ‘(‘ paramList ‘)’

      ```
                   | ID ‘(‘ error ‘)’  {….}
      ```
- Recover from syntax errors by discarding the tokens after an error until it reaches the valid token. In this way the parser can continue it's job even if there are some syntax errors in the code.

Using Bison With Flex

- Bison and flex are obviously designed to work together
- Bison produces a driver program called yylex() (actually it’s included in the lex library -ll)
  - #include “lex.yy.c” in the last part of bison specification
  - this gives the program yylex access to bisons’ token names

Using Bison with Flex (2)

- Thus do the following:
  - % flex scanner.l
  - % bison parser.y
  - % cc y.tab.c -ly -ll
- This will produce an a.out which is a parser with an integrated scanner included

Laboratory Assignment 3

Parser Generations

- Finnish a parser specification given in a *parser.y* bison file, by adding rules for expressions, conditions and function definitions, ...

Functions

```
function : funcnamedecl parameters ‘:’ type variables functions block ‘;’
{
    // Set the return type of the function
    // Set the function body
    // Set current function to point to the parent again
} ;

funcnamedecl : FUNCTION id
{
    // Check if the function is already defined, report error if so
    // Create a new function information and set its parent to current function
    // Link the newly created function information to the current function
    // Set the new function information to be current function
}  ;
```

Expressions

- For precedence and associativity you can factorize the rules for expressions …

  or

- you can specify precedence and associativy at the top of the Bison specification file, in the *Bison Declarations* section. Read more about this in the Bison reference(s).

Expressions (2)

- Example with factoring:

```
expression : expression '+' term
{
 // If any of the sub-expressions is NULL, set $$ to NULL
// Create a new Plus node but IntegerToReal casting might be needed
}
|
...
```

Conditions

- Example: a==b || a<c
- For precedence and associativity you can factorize the rules for conditions …

  or
- you can specify precedence and associativy at the top of the Bison specification file, in the *Bison Declarations* section. Read more about this in the Bison reference(s).

Laboratory Assignment 4

Intermediate Code

- Is closer to machine code without being machine dependent.
- Can handle temporary variables.
- Means higher portability, intermediate code can easier be expanded to assembly code.
- Offers the possibility of performing code optimizations such as register allocation.

Intermediate Code (2)

- Why do we use intermediate languages?
- Retargeting - build a compiler for a new machine by attaching a new code generator to an existing front-end and middle-part
- Optimization - reuse intermediate code optimizers in compilers for different languages and different machines
- Code generation - for different source languages can be combined

Intermediate Languages

- Various types of intermediate code are:
    - Infix notation
    - Postfix notation
    - Three address code
        - Triples
        - Quadruples

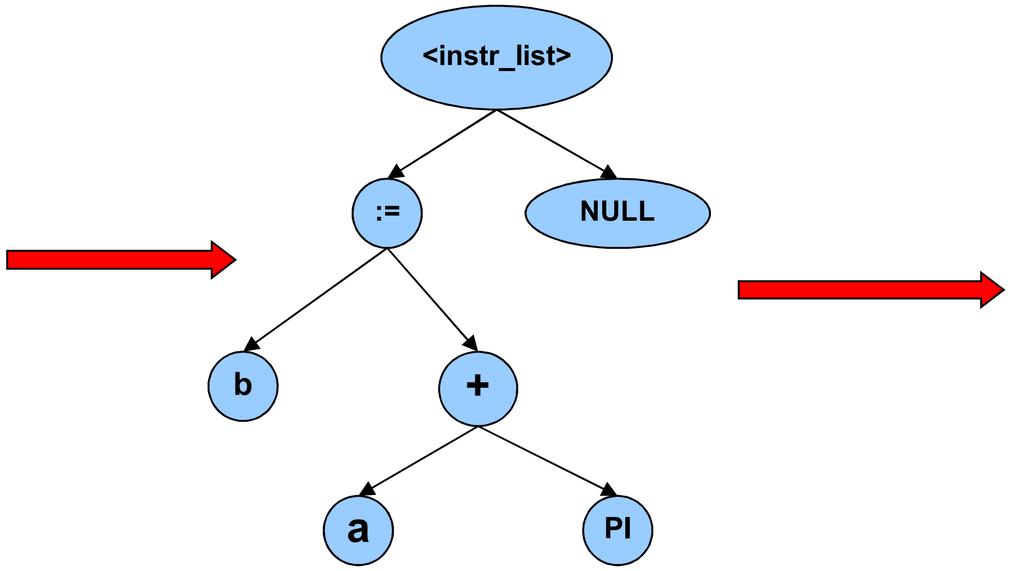
Quadruples

- You will use quadruples as intermediate language where an instruction has four fields:

```
operator     operand1   operand2   result
```

Generation of Intermediate Code

```
program example;
const
  PI = 3.14159;
var
  a : real;
  b : real;
begin
  b := a + PI;
end.
```



```
q_rplus    A   PI  $1
q_rassign  $1  -   B
q_labl     4   -   -
```

Quadruples

(A + B) * (C + D) - E

operator	operand1	operand2	result
+	A	B	T1
+	C	D	T2
*	T1	T2	T3
-	T3	E	T4

Intermediate Code Generation

- The purpose of this assignment is to learn how abstract syntax trees can be translated into intermediate code.
- You are to finish a generator for intermediate code (quadruples) by adding rules for some language constructs.
- You will work in the file *codegen.cc*.

Binary Operations

- In function *BinaryGenerateCode*:
  - Generate code for left expression and right expression.
  - Generate either a *realop* or *intop* quadruple
    - Type of the result is the same as the type of the operands
    - You can use *currentFunction->TemporaryVariable*

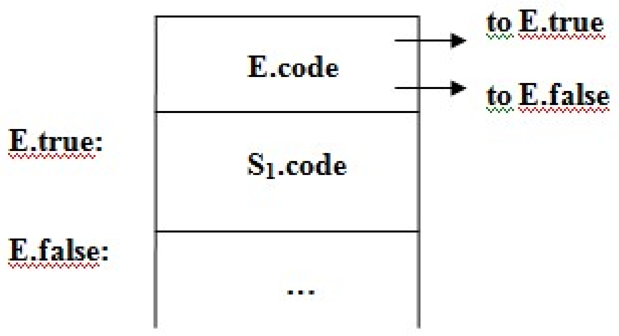
Array References

- The absolute address is computed as follows:
  - *absAdr = baseAdr + arrayTypeSize * index*

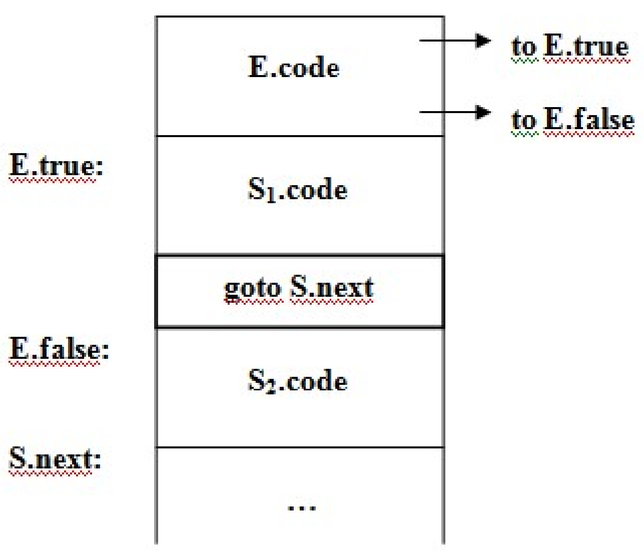
- Generate code for the index expression
- You must then compute the absolute memory address
  - You will have to create several temporary variables (of integer type) for intermediate storage
  - Generate a quadruple *iaddr* with *id* variable as input to get the base address
  - Create a quadruple for loading the size of the type in question to a temporary variable
  - Then generate *imul* and *iadd* quadruples
  - Finally generate either a *istore* or *rstore* quadruple

IF Statement

- **S → if E then S_1**
- **S → if E then S_1 else S_2**



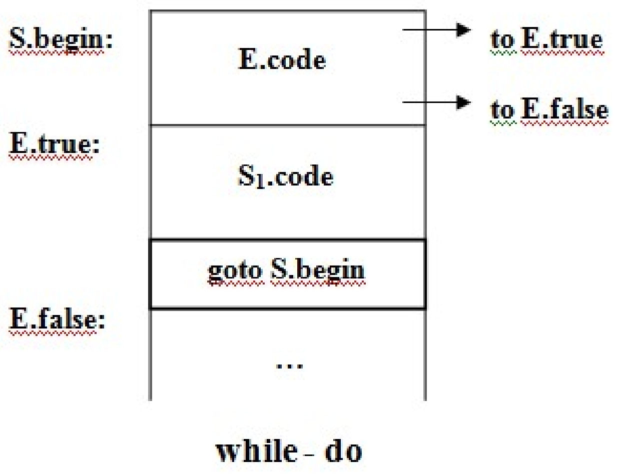
if - then



if – then - else

WHILE Statement

- **S → while E do S_1**



Questions?