TDDD55- Compilers and Interpreters
Lesson 3

Zeinab Ganjei (zeinab.ganjei@liu.se)

Department of Computer and Information Science
Linköping University

Laboratory Assignment 3

Parser Generation

- 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 the current function
} ;
```

Expressions

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

Expressions (2)

- Example with factoring:

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

Laboratory Assignment 4

Intermediate code

Intermediate Code

- Closer to machine code, but not machine specific
- 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

- 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

- Infix notation
- Postfix notation
- Three address code
  - Triples
  - Quadruples

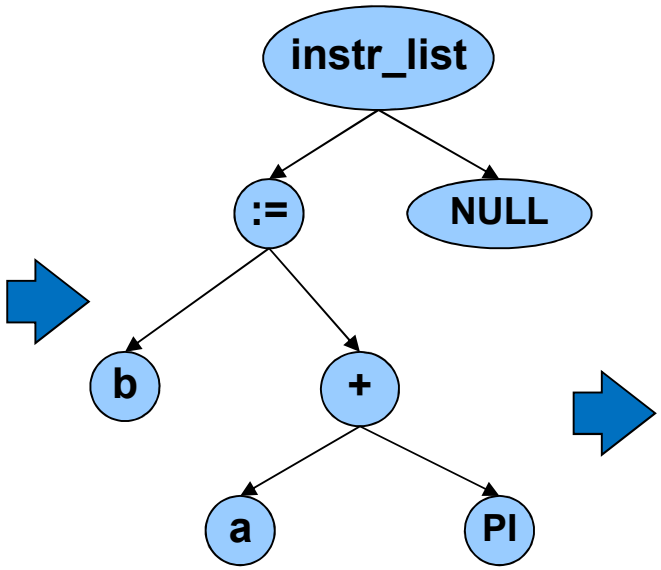
Quadruples

- 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

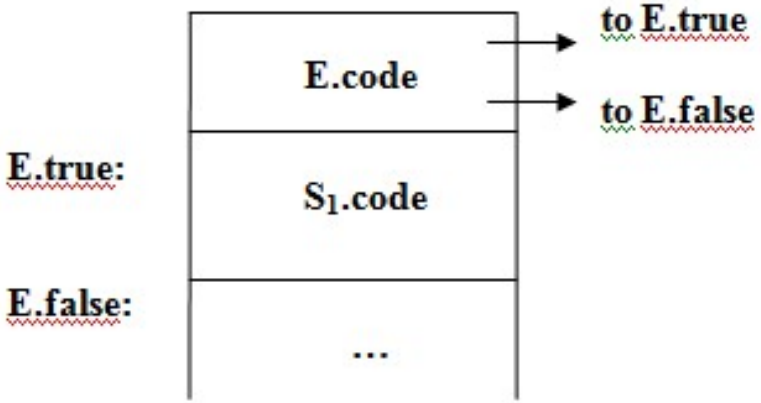
- In function *BinaryGenerateCode*:
  - Generate code for left expression and right expression.
  - Generate either a *realop* or *intop* quadruple
    - Type of the result is the same as the type of the operands
    - You can use *currentFunction->TemporaryVariable*

Array References

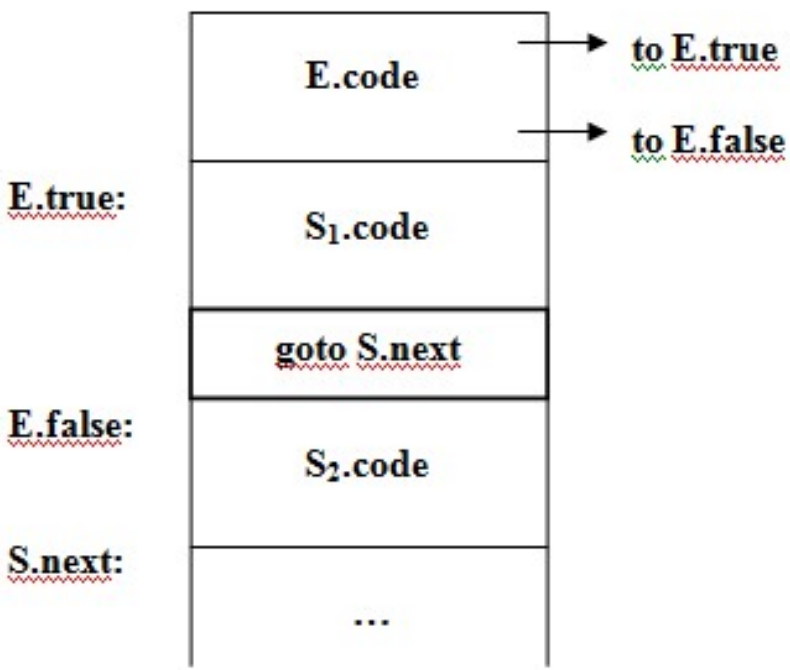
- The absolute address is computed as follows:
  - *absAdr = baseAdr + arrayTypeSize * index*
- Generate code for the index expression
- You must then compute the absolute address
  - You will have to create several temporary variables (of integer type) for intermediate storage
  - Generate a quadruple *iaddr* with *id* variable as input for getting 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**