University of Mosul
College Computer Science & Mathematics
Department of Computer Science

Compiler 2

Lecture 5: TYPE checking and Intermediate Code Generation

BY: Assist Prof. Dr. Ielaf O Abdul Majjed

Semantic Analysis (TYPE checking)

A semantic analyzer checks the source program for semantic errors. Type-checking is an important part of semantic analyzer.
Type checking is the process of verifying and enforcing constraints of types in values and attempts to catch programming errors based on the theory of types.
Two types of semantic Checks are performed within this phase these are:-

1. **Static Semantic Checks are performed at compile time** like:-
   - ❖ Type checking.
   - ❖ Every variable is declared before used.
   - ❖ Identifiers are used in appropriate contexts.
   - ❖ Check labels
2. **Dynamic Semantic Checks are performed at run time**, and the compiler produces code that performs these checks:-
   - ❖ Array subscript values are within bounds.
   - ❖ Arithmetic errors, e.g. division by zero.
   - ❖ A variable is used but hasn't been initialized.

Three kinds of languages:

1- **Statically**(typed: All or almost all checking of types is done as part of compilation (C, Java)
2- **Dynamically** (typed: Almost all checking of types is done as part of program execution (Scheme)
3- **Un-typed**: No type checking (machine code).

NOTE: Some programming languages such as C will combine both static and dynamic typing i.e. some types are checked before execution while others during execution

The design of type checker depends on:

1. Syntactic structure of language constructor.
2. The type expression of language.
3. The rules of the assigning types to construct.

Type Expression and Type Systems

Type Expression

The type of a language construct will be denoted by a type expression. A type expression is either a basic type or is formed by applying an operator called a type constructor to other type expressions.

1. **Basic type**
   - Integer: 7, 34, 909.
   - Floating point: 5.34, 123, 87.
   - Character: a, A.
   - Boolean: not, and, or, xor.
2. **Type constructor**
   - Arrays: If T is a type expression, then array (I, T) is a type expression denoting the type of an array with elements of type T and index set I.
   - Products: If T1 and T2 are type expressions, then their Cartesian product T1×T2 is a type expression.
   - Records: The type of a record is in a sense the product of the types of its fields. The difference between a record and a product is that the fields of a record have names.
   - Pointers: If T is a type expression, then pointer (T) is a type expression denoting the type pointer to an object of type T.
   - Functions: Functions take values in some domain and map them into value in some range.

Type System

Collection of rules for assigning types expression. In most languages, types system are:

1. Basic types are the atomic types with no internal structure as far as the programmer is concerned (int, char, float,….).
2. Constructed types are arrays, records, and sets. In addition, pointers and functions can also be treated as constructed types.
3. Type Equivalence:

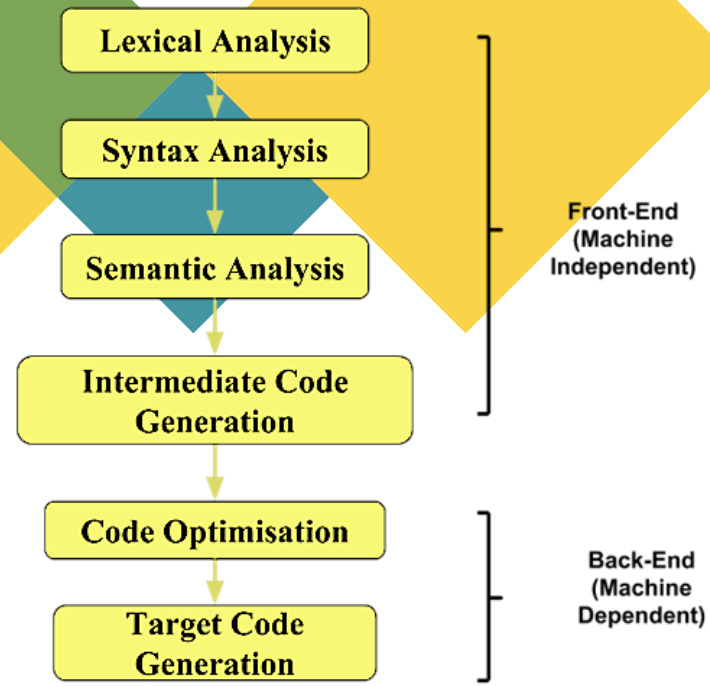
- Name equivalence: Types are equivalence only when they have the same name.
- Structural equivalence: Types are equivalence when they have the same structure.
- Example: In C uses structural equivalence for structs and name equivalence for arrays/pointers.

INTERMEDIATE CODE GENERATION

In the analysis-synthesis model of a compiler, the front end of a compiler translates a source program into an independent intermediate code, then the back end of the compiler uses this intermediate code to generate the target code (which can be understood by the machine).

The benefits of using machine independent intermediate code are:

- If a compiler translates the source language to its target machine language without having the ability to generate intermediate code, so for each new machine a full native compiler is required.
- The intermediate code eliminates the need for a complete new compiler for every single machine by keeping the parsing part the same for all compilers.
- The second part of the compiler, the synthesis, is modified depending on the target machine.
- It becomes easier to apply source code changes to improve code performance by applying code optimization techniques on intermediate code.



If we generate machine code directly from source code then for n target machine we will have n optimizers and n code generators but if we will have a machine independent intermediate code, we will have only one optimizer. Intermediate code can be either language specific (e.g., Bytecode for Java) or language independent (three-address code).

The following are commonly used intermediate code representation:

1. Postfix Notation

The ordinary (infix) way of writing the sum of a and b is with operator in the middle: a + b

The postfix notation for the same expression places the operator at the right end as ab +. In general, if e1 and e2 are any postfix expressions, and + is any binary operator, the result of applying + to the values denoted by e1 and e2 is postfix notation by e1e2 +. No parentheses are needed in postfix notation because the position and arity (number of arguments) of the operators permit only one way to decode a postfix expression. In postfix notation the operator follows the operand.

Example

The postfix representation of the expression (a – b) * (c + d) + (a – b) is: ab – cd + *ab -+

2. Three-Address Code

A statement involving no more than three references (two for operands and one for result) is known as three address statement. A sequence of three address statements is known as three address code. Three address statement is of the form

x = y op z where x, y, z will have address (memory location).

Sometimes a statement might contain less than three references but it is still called three address statement.

Example – The three address code for the expression a + b * c + d :

T 1 = b * c

T 2 = a + T 1

T 3 = T 2 + d

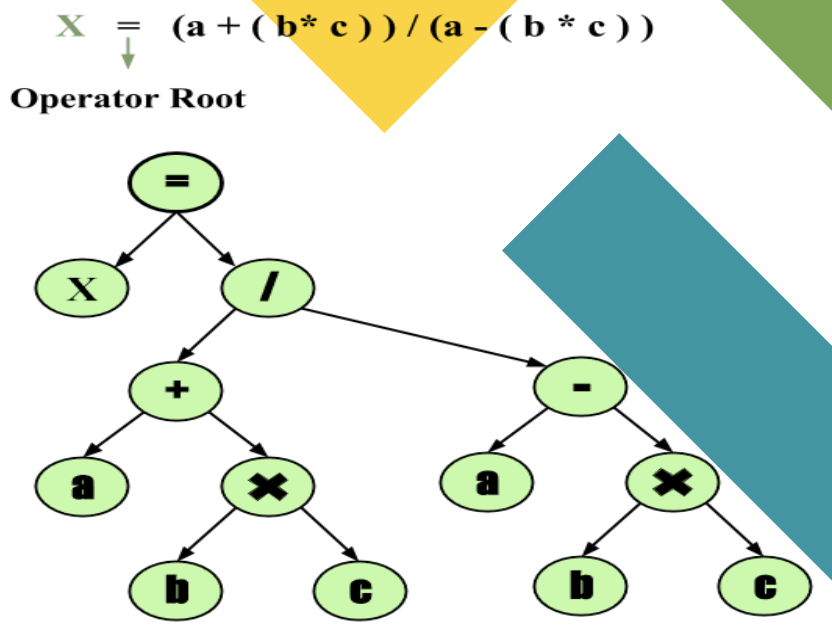
T 1 , T 2 , T 3 are temporary variables.

3. Syntax Tree

Syntax tree is nothing more than condensed form of a parse tree. The operator and keyword nodes of the parse tree are moved to their parents and a chain of single productions is replaced by single link in syntax tree the internal nodes are operators and child nodes are operands. To form syntax tree put parentheses in the expression, this way it's easy to recognize which operand should come first.

Example –

x = (a + b * c) / (a – b * c)



Issues in the design of a code generator

Code generator converts the intermediate representation of source code into a form that can be readily executed by the machine. A code generator is expected to generate the correct code. Designing of code generator should be done in such a way so that it can be easily implemented, tested and maintained.

1.Input to code generator

The input to code generator is the intermediate code generated by the front end, along with information in the symbol table that determines the run-time addresses of the data-objects denoted by the names in the intermediate representation. Intermediate codes may be represented mostly in quadruples, triples, indirect triples, Postfix notation, syntax trees, DAG's, etc. The code generation phase just proceeds on an assumption that the input are free from all of syntactic and state semantic errors, the necessary type checking has taken place and the type-conversion operators have been inserted wherever necessary.

2. Target program

The target program is the output of the code generator. The output may be absolute machine language, relocatable machine language, assembly language.

1. Absolute machine language as output has advantages that it can be placed in a fixed memory location and can be immediately executed.
2. Relocatable machine language as an output allows subprograms and subroutines to be compiled separately. Relocatable object modules can be linked together and loaded by linking loader. But there is added expense of linking and loading.
3. Assembly language as output makes the code generation easier. We can generate symbolic instructions and use macro-facilities of assembler in generating code. And we need an additional assembly step after code generation.

3. Memory Management:

Mapping the names in the source program to the addresses of data objects is done by the front end and the code generator. A name in the three address statements refers to the symbol table entry for name. Then from the symbol table entry, a relative address can be determined for the name.

4. Instruction selection:

Selecting the best instructions will improve the efficiency of the program. It includes the instructions that should be complete and uniform. Instruction speeds and machine idioms also plays a major role when efficiency is considered. But if we do not care about the efficiency of the target program then instruction selection is straight-forward.

For example, the respective three-address statements would be translated into the latter code sequence as shown below:

```
P:=Q+R
S:=P+T

 MOV Q, R0
ADD R, R0
MOV R0, P
MOV P, R0
ADD T, R0
MOV R0, S
```

Here the fourth statement is redundant as the value of the P is loaded again in that statement that just has been stored in the previous statement. It leads to an inefficient code sequence. A given intermediate representation can be translated into many code sequences, with significant cost differences between the different implementations. A prior knowledge of instruction cost is needed in order to design good sequences, but accurate cost information is difficult to predict.

5. Register allocation issues:

Use of registers make the computations faster in comparison to that of memory, so efficient utilization of registers is important. The use of registers are subdivided into two sub-problems:

1. During **Register allocation –** we select only those set of variables that will reside in the registers at each point in the program.
2. During a subsequent **Register assignment** phase, the specific register is picked to access the variable.

As the number of variables increases, the optimal assignment of registers to variables becomes difficult. Mathematically, this problem becomes NP-complete. Certain machine requires register pairs consist of an even and next odd-numbered register. For example M a, b

These types of multiplicative instruction involve register pairs where the multiplicand is an even register and b, the multiplier is the odd register of the even/odd register pair.

6. Evaluation order:

The code generator decides the order in which the instruction will be executed. The order of computations affects the efficiency of the target code. Among many computational orders, some will require only fewer registers to hold the intermediate results. However, picking the best order in the general case is a difficult NP-complete problem.

7. Approaches to code generation issues:

Code generator must always generate the correct code. It is essential because of the number of special cases that a code generator might face. Some of the design goals of code generator are:

- Correct
- Easily maintainable
- Testable
- Efficient