**Syntax Analysis**

• Syntax analyzers are based directly on the grammars.

• Lexical and syntax analyzers are needed in numerous situations outside compiler design including o program listing formatters o programs that compute the complexity of programs o programs that must analyze and react to the contents of a configuration . Introduction Lexical and Syntax Analysis are the first two phases of compilation as shown below. Lexical Analysis (Scanner) Syntax Analysis (Parser) characters tokens abstract syntax tree. Lexical and Syntax Analysis Languages are designed for both phases

• For characters, we have the language of regular expressions to recognize tokens.

• For tokens, we have context free grammars to recognize syntactically correct programs. Reasons for separating lexical analysis from syntax analysis are:

1. Simplicity – Techniques for lexical analysis are less complex that those required for syntax analysis, so the lexical-analysis process can be simpler if it separate. Also, removing the low-level details of lexical analysis from the syntax analyze makes the syntax analyzer both smaller and cleaner.

2. Efficiency – Although it pays to optimize the lexical analyzer, because lexical analysis requires a significant portion of total compilation time, it is not fruitful to optimize the syntax analyzer. Separation facilitates this selective optimization.

3. Portability – Because the lexical analyzer reads input program files and often includes buffering of that input, it is somewhat platform dependent. However, the syntax analyzer can be platform independent. It is always a good practice to isolate machine dependent parts of any software system.

**Immediate Code Generation**

During the translation of a source program into the object code for a target machine, a compiler may generate a middle-level language code, which is known as **intermediate code** or **intermediate text**. The complexity of this code lies between the source language code and the object code. The intermediate code can be represented in the form of postfix notation, syntax tree, directed acyclic graph (DAG), three-address code, quadruples, and triples

**Utility of Intermediate Code Generation:**

1. Suppose we have n-source languages and m-Target languages. Without Intermediate code. We will change each source language into target language directly.

2. So, for each source-target pair we will need a compiler. Hence we will require (n\*m) Compilers, one for each pair. If we Use Intermediate code. We will require n-Compilers to convert each source language into Intermediate code and mCompilers to convert Intermediate code into m-target languages. Thus We require only (n+m) Compilers.

**Different Types of Intermediate codes**

Intermediate code must be easy to produce and easy to translate to machine code

 It is a sort of universal assembly language.

 It should not contain any machine-specific parameters (registers, addresses, etc.)

The type of intermediate code deployed is based on the application like Quadruples, triples, indirect triples, abstract syntax trees are the classical forms used for machine-independent optimizations and machine code generation.

Static Single Assignment form (SSA) is a recent form and enables more effective optimizations.

Conditional constant propagation and global value numbering are more effective on SSA Program Dependence Graph (PDG) is useful in automatic parallelization, instruction scheduling, and software pipelining

**Three Address Code**

Instructions are very simple

Examples: a = b + c, x = -y, if a > b goto L1

LHS is the target and the RHS has at most two sources and one operator

RHS sources can be either variables or constants

Three-address code is a generic form and can be implemented as quadruples, triples, indirect triples, tree or DAG.

Example: The three-address code for a+b\*c-d/(b\*c) is below

● t1 = b\*c

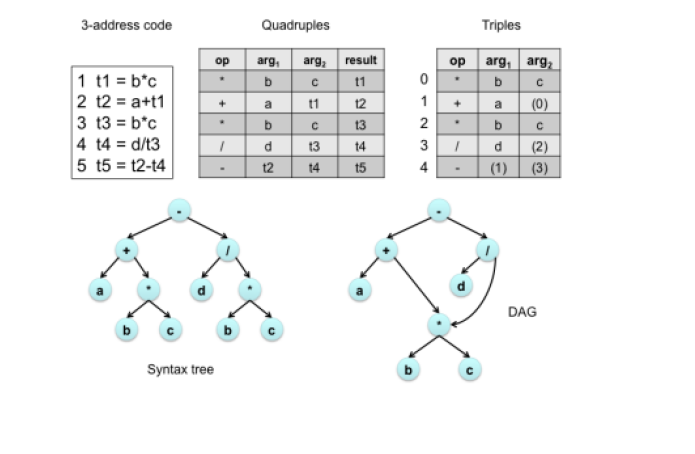
● t2 = a+t1

● t3 = b\*c

● t4 = d/t3

● t5 = t2-t4

**Implementation of three address code**



Translate the following expression to quadruple, triple and indirect triple-

a = b x – c + b x – c

**Solution-**

Three Address Code for the given expression is-

Three Address Code for the given expression is-

T1 = uminus c

T2 = b x T1

T3 = uminus c

T4 = b x T3

T5 = T2 + T4

a = T5

### ****Quadruple Representation-****

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Location** | **Op** | **Arg1** | **Arg2** | **Result** |
| (1) | uminus | c |  | T1 |
| (2) | x | b | T1 | T2 |
| (3) | uminus | c |  | T3 |
| (4) | x | b | T3 | T4 |
| (5) | + | T2 | T4 | T5 |
| (6) | = | T5 |  | a |

### ****Triple Representation-****

|  |  |  |  |
| --- | --- | --- | --- |
| **Location** | **Op** | **Arg1** | **Arg2** |
| (1) | uminus | c |  |
| (2) | x | b | (1) |
| (3) | uminus | c |  |
| (4) | x | b | (3) |
| (5) | + | (2) | (4) |
| (6) | = | a | (5) |

### ****Indirect Triple Representation-****

|  |  |
| --- | --- |
|  | **Statement** |
| 35 | (1) |
| 36 | (2) |
| 37 | (3) |
| 38 | (4) |
| 39 | (5) |
| 40 | (6) |

|  |  |  |  |
| --- | --- | --- | --- |
| **Location** | **Op** | **Arg1** | **Arg2** |
| (1) | uminus | c |  |
| (2) | x | b | (1) |
| (3) | uminus | c |  |
| (4) | x | b | (3) |
| (5) | + | (2) | (4) |
| (6) | = | a | (5) |

Online Notes

<https://people.cs.vt.edu/prsardar/classes/cs3304-Spr19/lectures/CS3304-9-LanguageSyntax-2.pdf>

<https://www2.cs.sfu.ca/CourseCentral/383/dma/notes/lecture_sep29.pdf>

<https://www.cs.york.ac.uk/fp/lsa/lectures/slideshow1.pdf>

<https://nptel.ac.in/content/storage2/courses/106108113/module5/Lecture17.pdf>

<https://people.montefiore.uliege.be/geurts/Cours/compil/2012/05-intermediatecode-2012-2013.pdf>

<https://www.tutorialspoint.com/compiler_design/pdf/compiler_design_intermediate_code_generations.pdf>

Video links

<https://www.youtube.com/watch?v=BgNdtk9h8Ok>

<https://www.youtube.com/watch?v=LjB7ffXWzD0>

<https://www.youtube.com/watch?v=WccZQSERfCM>

<https://www.youtube.com/watch?v=j-bLeUysUiE>

<https://www.youtube.com/watch?v=SfRCkAoahuU>

<https://www.youtube.com/watch?v=yMUw2MxzNec>