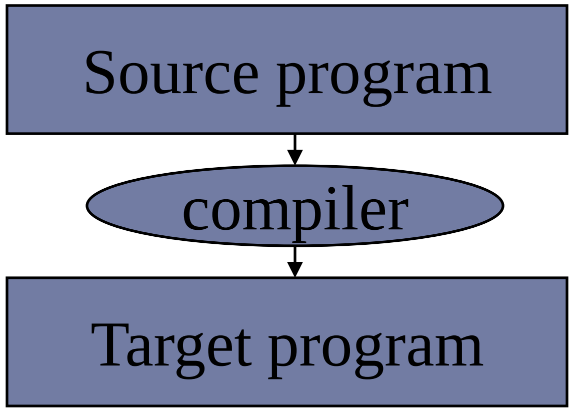
Introduction to Compiler

Translator

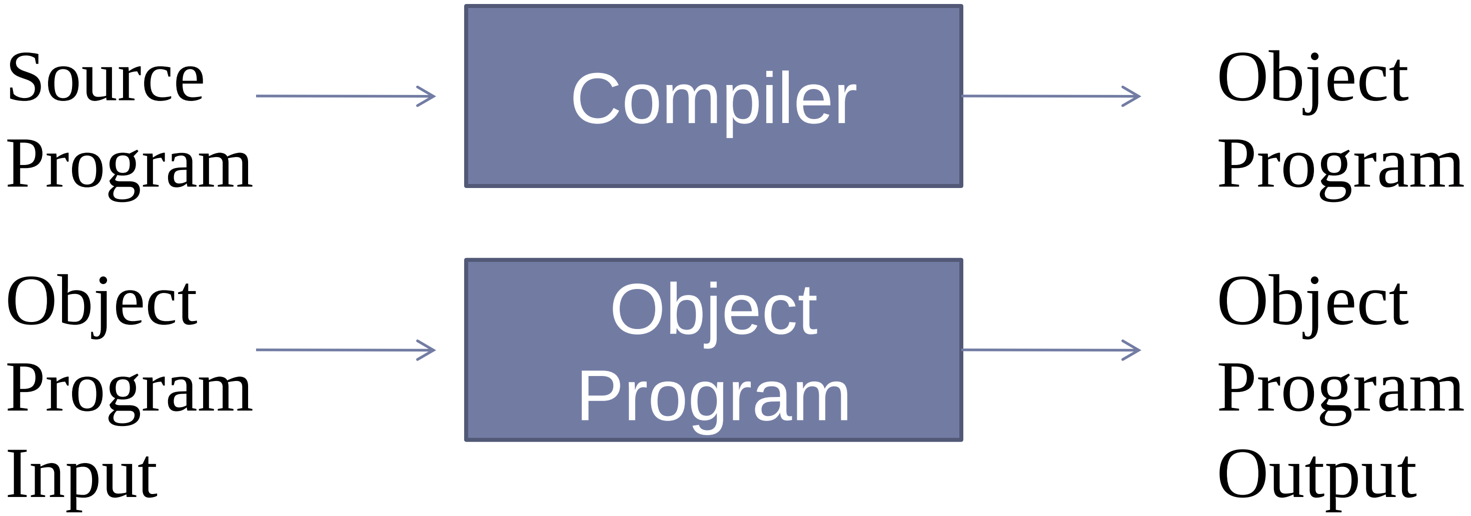
- A translator is a program that takes a program as input written in one programming language(the source language) and produces as output a program in another language(the object or target language).

What is a compiler?

- A program that accepts as input a program text in a certain language and produces as output a program text in another language, while preserving the meaning of that text (Grune *et al*, 2000).
- A program that reads a program written in one language (source language) and translates it into an equivalent program in another language (target language) (Aho *et al*)



- Executing a program written in a high-level programming language is basically two step process.
  - The Source Program must first be compiled i.e. translated to the object program.
  - The resulting object program is loaded into memory and executed.



What is an interpreter?

- A program that transforms a programming language into a simplified language, called intermediate code, which can be directly executed using a program called an interpreter.
- Interpreters are often smaller than compilers and facilitate the implementation of complex programming language constructs.
- The execution time of an interpreted program is usually slower than that of a corresponding compiled object program.

Examples

- C is typically compiled
- Lisp is typically interpreted

Assembler

- If the source language is assembly language and the target language is machine language then the translator is called assembler.

Preprocessor

- Preprocessors are those translators that convert program in one high-level language into equivalent programs in another high-level language.
- Example: There are many FORTRANN preprocessors that map “structured” versions of FORTRAN into conventional FORTRAN.

Why do we need translators?

- With machine language we must communicate directly with computer in terms of bits, registers and very primitive machine operations.
- Machine language program is a sequence of 0’s and 1’s.
- Programming a complex algorithm in such a language is terribly tedious and fraught with opportunities for mistakes.
- The main disadvantage of machine language coding is that all operations and operands must be specified in numeric code.
- Not only machine language programs cryptic, but it also may be impossible to modify in a convenient manner.

Symbolic Assembly Language

- Because of the difficulties with machine language programming, a host of “higher level” languages have been invented to enable the programmer to code in a way that resembles his own thought process rather than the elementary steps of a computer.
- The most immediate step away from the machine language is Symbolic Assembly language.
- In this language programmer uses mnemonic names for both operation codes and data Addresses.
- Thus programmer could write ADD X, Y in assembly language instead of 0110 001110 010101 in machine language.
- A computer can not execute a program written in assembly language. That program has to be first translated to machine language which the computer can understand. The program that performs this translation is called Assembler.

Macros

- A Macro statement translates a code into a sequence of assembly language statements and perhaps other macro statements before being translated into machine code.
- Thus a macro facility is a text replacement capability.
- There are two aspects of macros:
  - Definition
  - Use
- Example:

```
MACRO   ADD2      X,Y
         LOAD      Y
         ADD        X
         STORE    Y
ENDMACRO
```

- We assume that the machine has only one register.

Macros (contd…)

- Having defined ADD2 in this way, we can then use it as ordinary assembly language code.
- For Example, if the statement ADD2 A, B is encountered somewhere after the definition of ADD2, we have a macro use.
- Here, the macro processor substitutes for ADD2 A,B three statements which form the definition of ADD2, but with the actual parameters A and B replacing the formal parameters X and y, respectively. i.e. ADD2 A,B is translated to

```
      LOAD      B
      ADD       A
      STORE     B
```

High Level Languages

- There are some drawbacks of Assembly programs
  - The programmer must still know the details of how a specific program operates.
  - He must mentally translate complex operations and data structures into sequence of low-level operations which use only the primitive data types that machine language provides.
  - The programmer must be concerned with how and where the data is represented within the machine.
- To avoid the above problems, high level languages were developed.
- A high level language allows a programmer to express algorithms in a more natural notation that avoids many of the details of how a specific computer functions.
- A high level programming language makes the programming task simpler.

High Level Languages (contd..)

- We need a program to translate the high language code into a a language that the machine can understand.
- A Compiler translates a program written in high language into a program that a machine can understand.
- A compiler is more complex to write than assembler.
- Some compilers make use of an assembler as an appendage, with compiler producing assembly code, which is then assembled and loaded before being executed in the resulting machine language code.

Qualities of a Good Compiler

What qualities would you want in a compiler?

- generates correct code (first and foremost!)
- generates fast code
- conforms to the specifications of the input language
- copes with essentially arbitrary input size, variables, etc.
- compilation time (linearly)proportional to size of source
- good diagnostics
- consistent optimisations
- works well with the debugger

Principles of Compilation

The compiler must:

- preserve the meaning of the program being compiled.
- “improve” the source code in some way.

Other issues (depending on the setting):

- Speed (of compiled code)
- Space (size of compiled code)
- Feedback (information provided to the user)
- Debugging (transformations obscure the relationship source code vs target)
- Compilation time efficiency (fast or slow compiler?)

Uses of Compiler Technology

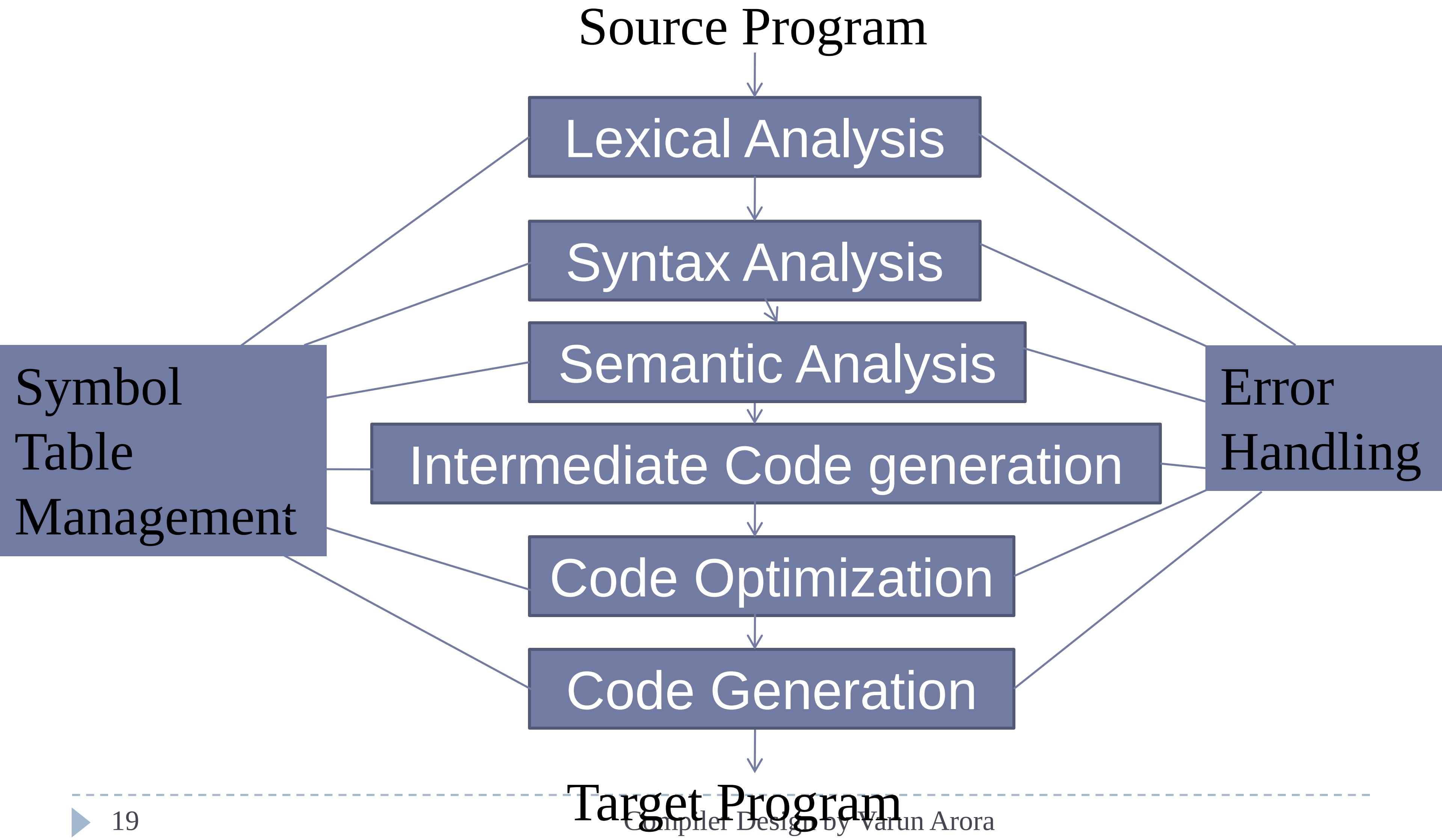
- Most common use: translate a high-level program to object code
- Optimizations for computer architectures:
- Automatic parallelisation or vectorisation
- Software productivity tools
- Security: Java VM uses compiler analysis to prove “safety” of Java code.
- Text formatters, just-in-time compilation for Java, power management, global distributed computing, …

Key: Ability to extract properties of a source program (analysis) and transform it to construct a target program (synthesis)

Structure of a Compiler

- Compilation process is partitioned into a series of sub processes called the phases.
- A phase is a logically cohesive operation that takes as input one representation of the source program and produces as output another representation.
- The structure of compilation process is shown in the fig.(next slide)

Structure of a Compiler(contd..)



Lexical Analysis (Scanning)

- Seperates the characters of the source language into groups that logically belong togather; these groups are called tokens.
- Usual tokens are keywords, such as DO or IF; identifiers like X or NUM, Operator Symbols such as <= or + and punctuation symbols such as paranthesis or commas.
- The output of the lexical analyzer is a stream of tokens, which is passed to the next phase, the syntax analyzer or parser.
- The lxical analyzer produces as output a token of the form
  <token-name, attribute-value>
- E.g.: **`a=b+c`** becomes <id,**`a`**> <=,> <id,**`b`**> <+,> <id,**`c`**>
- Needs to record each id attribute: keep a **symbol table**.
- Lexical analysis eliminates white space, etc…

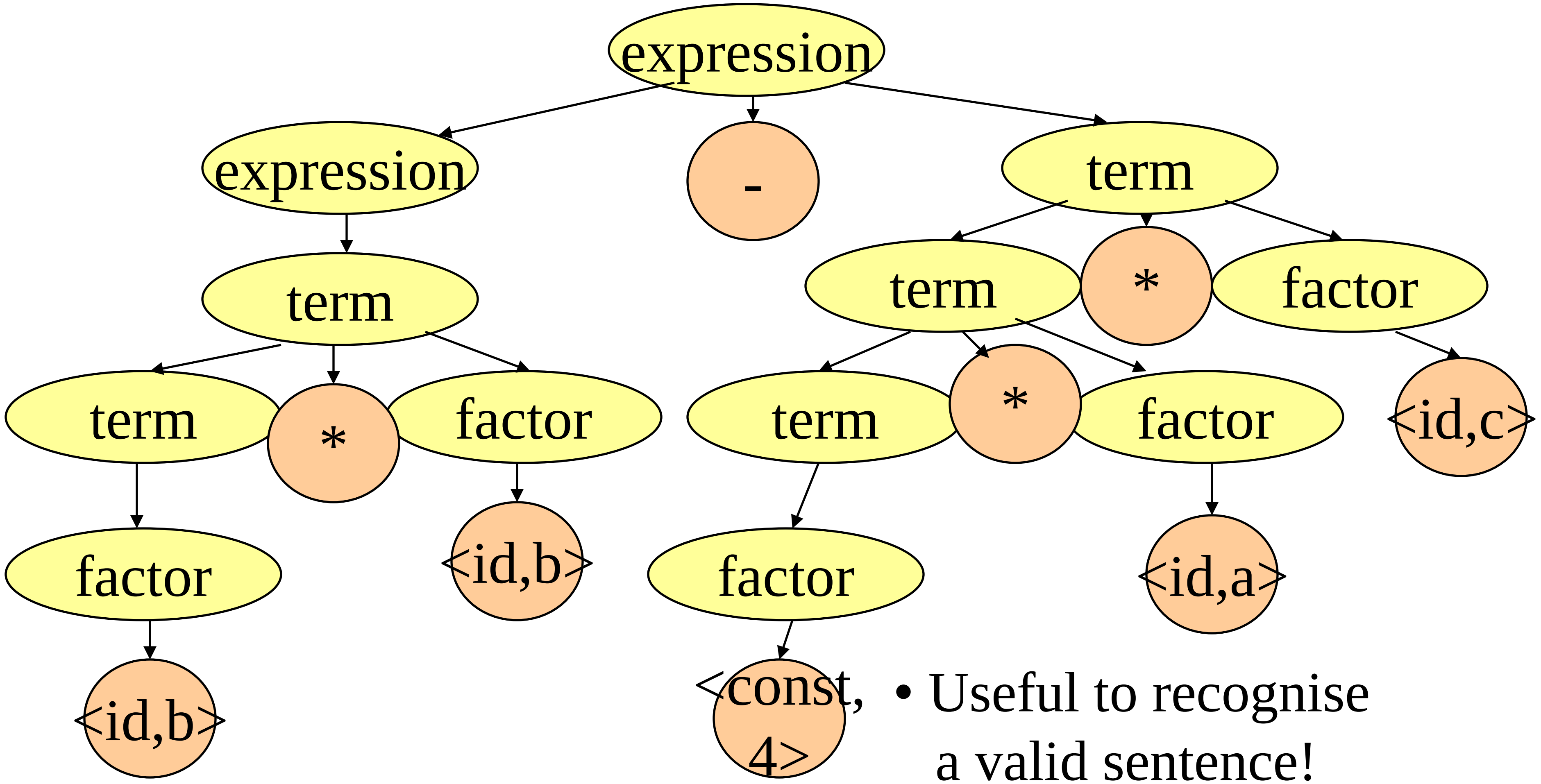
Syntax (or syntactic) Analysis (Parsing)

- Groups tokens together into syntactic structures.
- For example: three tokens representing A+B might be grouped into a syntactic structure called an expression.
- Expressions may further be combined to form statements.
- Often syntactic structure can be regarded as a tree whose leaves are tokens.
- Interior nodes of the tree represent string of tokens that logically belong together.
- Imposes a hierarchical structure on the token stream.
- This hierarchical structure is usually expressed by recursive rules.
- Context-free grammars formalise these recursive rules and guide syntax analysis.
- Example:

```
expression → expression '+' term | expression '-' term | term
term → term '*' factor | term '/' factor | factor
factor → identifier | constant | '(' expression ')'
```

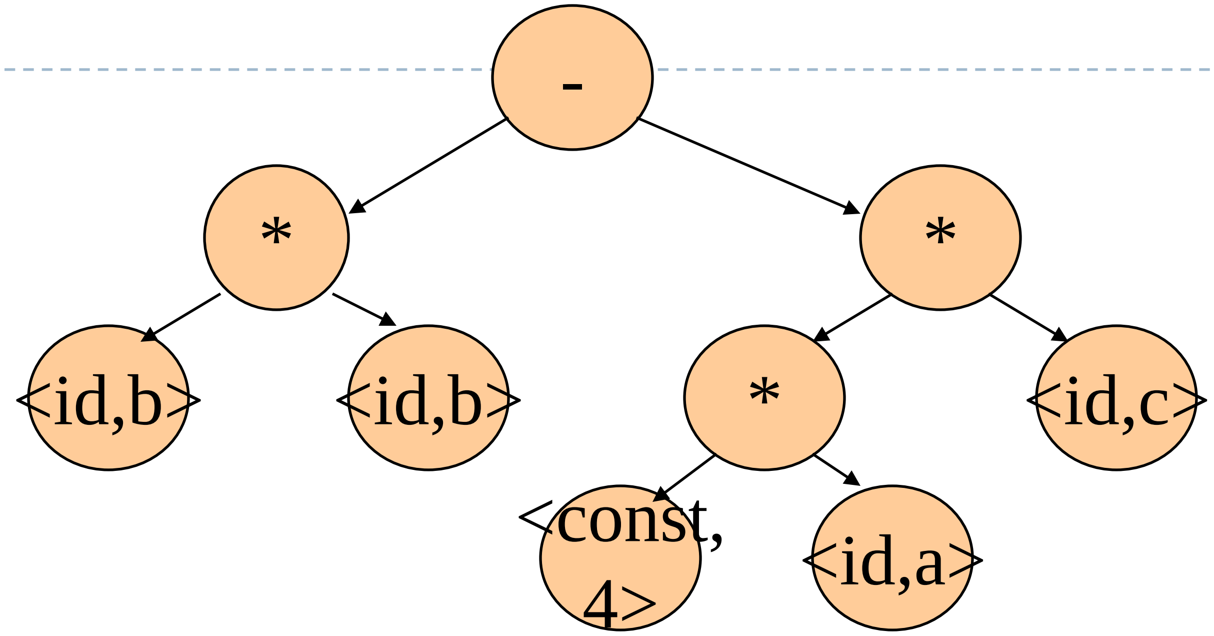
(this grammar defines simple algebraic expressions)

Parsing: parse tree for *b*b-4*a*c*



- Useful to recognise a valid sentence!
- Contains a lot of unneeded information!

AST for *b*b-4*a*c*



- An Abstract Syntax Tree (AST) is a more useful data structure for internal representation. It is a compressed version of the parse tree (summary of grammatical structure without details about its derivation)

Semantic Analysis (context handling)

- Uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition.
- Collects context (semantic) information, checks for semantic errors, and annotates nodes of the tree with the results.
- Examples:
  - type checking: report error if an operator is applied to an incompatible operand.
  - check flow-of-controls.
  - uniqueness or name-related checks.

Intermediate code generation

- Uses the structure produced by the syntax analyzer to create a stream of simple instructions.
- Translate language-specific constructs in the AST into more general constructs.
- A criterion for the level of “generality”: it should be straightforward to generate the target code from the intermediate representation chosen.

Code Optimisation

- It is an optional phase designed to improve the intermediate code so that the ultimate object program runs faster and/or takes less space.
- Its output is another intermediate code program that does the same job as original but saves time and space.

Code Generation Phase

- Produces the object code by deciding on memory locations for data, selecting code to access each datum, and selecting the registers in which each computation is to be done.

Table - Management or Bookkeeping

- Keeps the track of the names used by the program and records essential information about each, such as its type (integer, real etc.) . The data structure used to record this information is called symbol table.

Error Handler

- It is invoked when a flaw in the source program is detected.
- It must warn the programmer by issuing a diagnostic and adjust the information being passed from phase to phase so that each phase can proceed.

Conceptual Structure:two major phases

- **The portions of one or more phases are combined into a module called passes.**
- **A pass reads the source program or the output of previous pass , makes the transformations specified by its phases and writes output into an intermediate file, which may then be read by subsequent pass.**
- **The number of passes and the grouping of phases into the passes, are usually dictated by a variety of considerations to a perticular language or machine.**
- **The structure of the source language has a strong effect on the number of passes.**
- **Certain languages require at least two passes to generate code easily.**
- **The environment in which the compiler must operate can also affect the number of passes.**
- **A multi pass compiler can be made to use less space than single pass compiler, since the space occupied by the compiler program for one pass can be reused by the following pass.**
- ***A multi pass compiler is slower than a single pass compiler , because each pass reads and writes an intermediate file.***
- ***Thus compilers running on a computer with small memory would use several passes while, on a computer with a large RAM , a compiler with fewer passes will be possible.***

- **Front-end** performs the **analysis** of the source language:
  - Recognises legal and illegal programs and reports errors.
  - “understands” the input program and collects its semantics in an IR.
  - Produces IR(Intermediate Representation) and shapes the code for the back-end.
  - Much can be automated.
- **Back-end** does the target language **synthesis**:
  - Chooses instructions to implement each IR operation.
  - Translates IR into target code.
  - Needs to conform with system interfaces.
  - Automation has been less successful.

Cross Compiler

- A compiler is characterized by three languages: its source language, its object language and the language in which it is written.
- These languages may all be different.
- A compiler may run on one machine and produce object code for other machine.
- Such compiler is called Cross Compiler.
- Many minicomputer and microprocessor compilers are implemented in this way, they run on a bigger machine and produce object code for the smaller machine.

Bootstrapping (compilers)

- **Bootstrapping** is the technique for producing a self-compiling compiler — that is, compiler (or assembler) written in the source programming language that it intends to compile.
- An initial core version of the compiler (the *bootstrap compiler*) is generated in a different language (which could be assembly language); successive expanded versions of the compiler are developed using this minimal subset of the language.
- Many compilers for many programming languages are bootstrapped, including compilers for BASIC, ALGOL, C etc.