An Overview of a Compiler - Part 2

Y.N. Srikant

Department of Computer Science
Indian Institute of Science
Bangalore 560 012

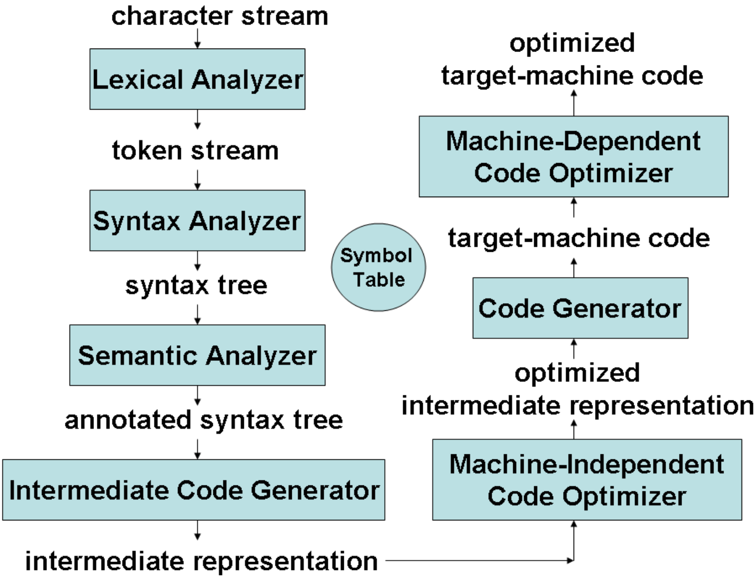
NPTEL Course on Compiler Design

Outline of the Lecture

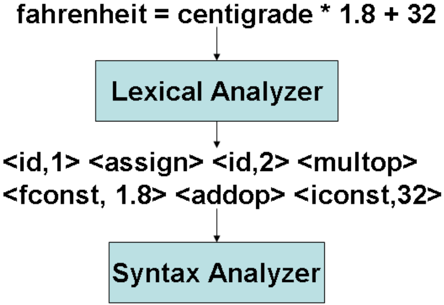
1. Compiler overview with block diagram
2. Lexical analysis with LEX
3. Parsing with YACC
4. Semantic analysis with attribute grammars
5. Intermediate code generation with syntax-directed translation
6. Code optimization examples

Topics 1 to 4 have been covered in Part I of the lecture

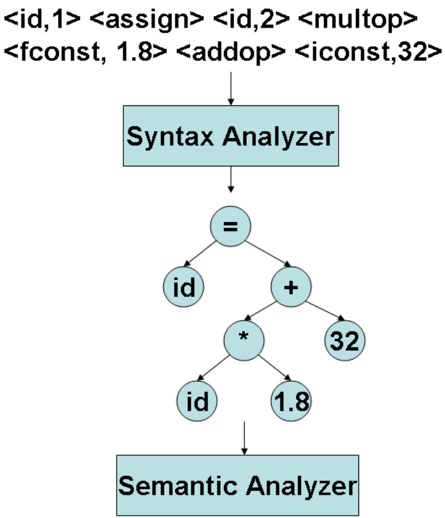
Compiler Overview



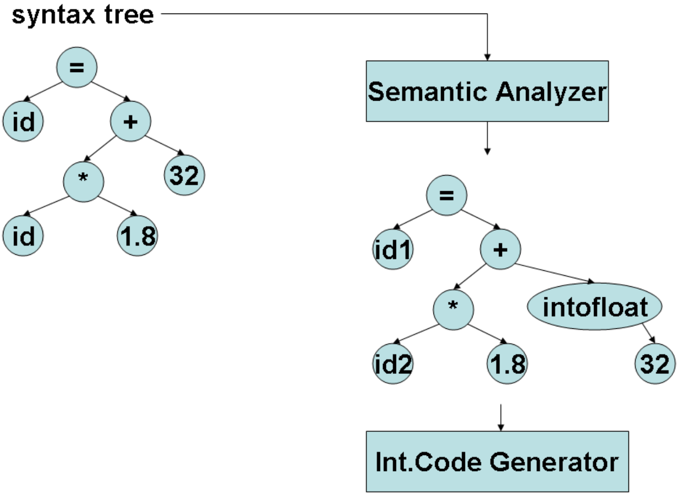
Translation Overview - Lexical Analysis



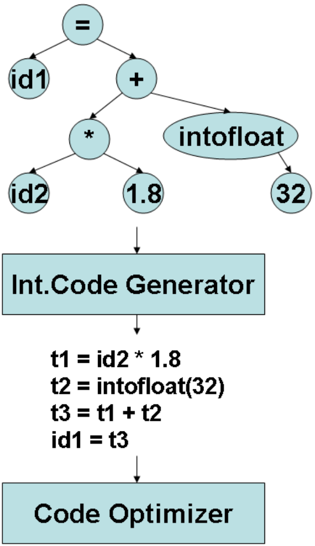
Translation Overview - Syntax Analysis



Translation Overview - Semantic Analysis



Translation Overview - Intermediate Code Generation



Intermediate Code Generation

- While generating machine code directly from source code is possible, it entails two problems
  - With m languages and n target machines, we need to write $m \times n$ compilers
  - The code optimizer which is one of the largest and very-difficult-to-write components of any compiler cannot be reused
- By converting source code to an intermediate code, a machine-independent code optimizer may be written
- Intermediate code must be easy to produce and easy to translate to machine code
  - A sort of universal assembly language
  - Should not contain any machine-specific parameters (registers, addresses, etc.)
- Usually produced during a traversal of the semantically validated syntax tree

Different Types of Intermediate Code

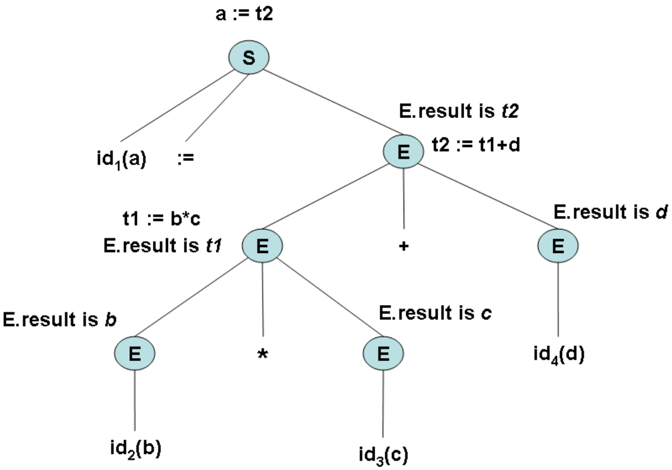
- The type of intermediate code deployed is based on the application
- Quadruples, triples, indirect triples, abstract syntax trees are the classical forms used for machine-independent optimizations and machine code generation
- Static Single Assignment form (SSA) is a recent form and enables more effective 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

Translation to produce Quadruples for Expressions

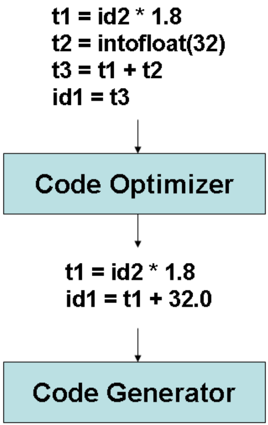
1. $S \rightarrow id := E$ $\{idptr := search(id.name);$
   $if\ idptr \neq NULL\ then\ gen(idptr\ ' :=' \ E.result\ else\ error\}$
2. $E \rightarrow E_1 + E_2$ $\{E.result := gentemp();$
   $gen(E.result\ ' :=' \ E_1.result\ ' +' \ E_2.result)\}$
3. $E \rightarrow E_1 * E_2$ $\{E.result := gentemp();$
   $gen(E.result\ ' :=' \ E_1.result\ ' *' \ E_2.result)\}$
4. $E \rightarrow -E_1$ $\{E.result := gentemp();$
   $gen(E.result\ ' :=' \ 'uminus' \ E_1.result)\}$
5. $E \rightarrow (\ E_1\ )$ $\{E.result := E_1.result\}$
6. $E \rightarrow id$ $\{idptr := search(id.name);$
   $if\ idptr \neq NULL\ then\ E.result := idptr\ else\ error\}$

- Names are stored in a symbol table; the routine *search*(*id.name*) gets a pointer to the name *id.name*
- *gentemp*() generates a temporary name, puts it in the symbol table, and returns a pointer to it

Quadruples for Expressions - An Example of Translation



Translation Overview - Code Optimization



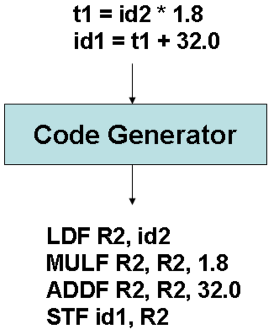
Machine-independent Code Optimization

- Intermediate code generation process introduces many inefficiencies
  - Extra copies of variables, using variables instead of constants, repeated evaluation of expressions, etc.
- Code optimization removes such inefficiencies and improves code
- Improvement may be time, space, or power consumption
- It changes the structure of programs, sometimes of beyond recognition
  - Inlines functions, unrolls loops, eliminates some programmer-defined variables, etc.
- Code optimization consists of a bunch of heuristics and percentage of improvement depends on programs (may be zero also)

Examples of Machine-Independant Optimizations

- Common sub-expression elimination
- Copy propagation
- Loop invariant code motion
- Partial redundancy elimination
- Induction variable elimination and strength reduction
- Code opimization needs information about the program
  - which expressions are being recomputed in a function?
  - which definitions reach a point?
- All such information is gathered through data-flow analysis

Translation Overview - Code Generation



Code Generation

- Converts intermediate code to machine code
- Each intermediate code instruction may result in many machine instructions or vice-cersa
- Must handle all aspects of machine architecture
  - Registers, pipelining, cache, multiple function units, etc.
- Generating efficient code is an NP-complete problem
  - Tree pattern matching-based strategies are among the best
  - Needs tree intermediate code
- Storage allocation decisions are made here
  - Register allocation and assignment are the most important problems

Machine-Dependent Optimizations

- Peephole optimizations
  - Analyze sequence of instructions in a small window (*peephole*) and using preset patterns, replace them with a more efficient sequence
  - Redundant instruction elimination
    e.g., replace the sequence [LD A,R1][ST R1,A] by [LD A,R1]
  - Eliminate “jump to jump” instructions
  - Use machine idioms (use INC instead of LD and ADD)
- Instruction scheduling (reordering) to eliminate pipeline interlocks and to increase parallelism
- Trace scheduling to increase the size of basic blocks and increase parallelism
- Software pipelining to increase parallelism in loops