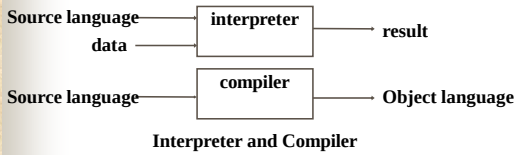
Compilers Principles and Practice review

Dong Min

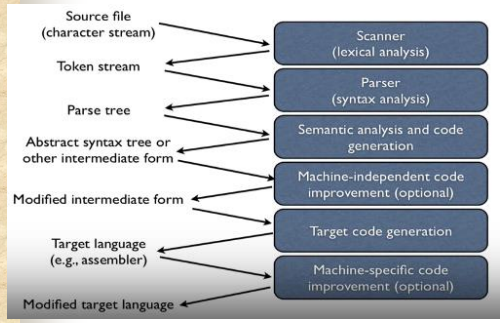
35866610@qq.com
1350706534

- **Interpreter and Compiler**
  - **The same point:They are all language implementing system**
  - **Differences:**
    - **Interpreter executes the source program during translation**
    - **Compiler generates object code that is executed after translation completes**



Interpreter and Compiler

The flow chart of the process of compiling



1 The Scanner

- **Scanner performs lexical analysis**
- **The Task of scanner:**

  Reading the source program as a file of characters and dividing it up into meaningful units called tokens
  - **Input:source program which is a stream of characters**
  - **Output:tokens**

- **Token**
  - **Each token is a sequence of characters that represents a unit of information in the source program.**

- Tokens fall into several categories :
  - Identifier:user-defined strings, usually consist of letters and numbers and begin with a letter
  - Keyword:fixed strings of letters, such as “if”,“while”
  - Number
  - Operator:such as arithmetic operator ‘+’ and ‘×’, “>= ”, “<= ”
  - Special symbol:such as ‘(‘ ‘)’ ‘;’
- Token is presented as (Kind,Value)

Example
a[index]=4+2

Source program in file

a	[	i	n	d	e	x	]	=	4	+	2

Token stream after lexical analysis

(identifier,a)	(left bracket,[)	(identifier,index)	(right bracket,])
(assignment,=)	(number,4)	(plus sign,+)	(number,2)

- Other operations performed by scanner
  - Enter identifiers into the symbol table
  - Enter literals in literal table, literals include
    - Numeric constants(3.14)
    - Quoted string of text(“hello”)

2 The Parser

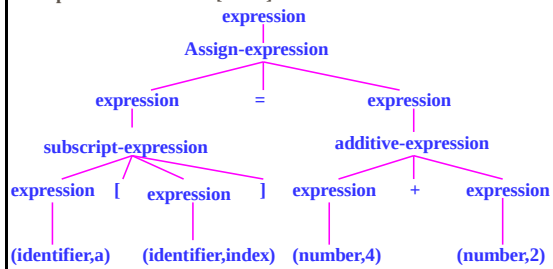
- Parser performs syntax analysis
- Task of parser

  The parser receives the source code in the form of tokens from the scanner and performs the syntax analysis ,which determines the structure of the program
  - Input:stream of tokens
  - Output:parse tree or syntax tree

- Parse tree

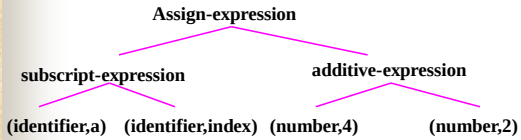
  Parse tree is a useful aid to visualizing the syntax of a program or program element

Example
The parse tree of code a[index]=4+2



- Internal nodes are labeled by the names of the structures they represent
- Leaves represent the sequence of tokens from the input

- **Abstract Syntax Tree(Syntax Tree):**
  A condensation of the information contained in the parse tree



3 The Semantic Analyzer

- **Semantic**
  - **Semantics of a program are its "meaning"**
  - **Static semantic: properties of a program that can be determined prior to execution**
  - **Dynamic semantic:properties of a program that can only be determined by execution**
- **Task of Semantic Analyzer**
  Analyze static semantic

4 The Source Code Optimizer

- **Optimization performed after semantic analysis that depend only on the source code**
  - **Some optimizations can be performed directly on the syntax tree**
  - **It is easier to optimize useing Intermediate Code**

- **Intermediate Code**
  - **A form of code representation intermediate between source code and object code**
  - **Intermediate codes have the following properties: structure is simple, meaning is clear, and it is easy to translate them to object code**
  - **For example**
    Three-address code: it contains the addresses of three locations in memory

- **IR(Intermediate Representation)**
  - **Any internal representation for the source code used by the compiler is called IR**
  - **The syntax tree and intermediate code are all IR**

5 The Code Generator

- **Task of Code Generator**
  - **The code generator takes the IR and generates code for the target machine**
  - **Usually use assembly language as target code**
  - **It is related to the properties of target machine: the number of registers, addressing mode, data representation and so on.**

Example

Target code for a[index]=6 in assembly language:

```
MOV R0,index
MUL R0,2
MOV R1,&a
ADD R1,R0
MOV *R1,6
```

These codes correspond to the following conventions on target machine:

- An integer occupies two bytes of memory
- &a is the address of a
- *R means indirect register addressing

6 The Target Code Optimizer

- Task of the Target Code Optimizer

  To improve the target code generated by the code generator, saving execution time and memory space
- This Optimization includes
  - Change addressing mode to improve performance
  - Replace slow instructions by faster ones
  - Eliminate redundant or unnecessary operations

Example

Target codes are:

```
MOV R0,index
MUL R0,2
MOV R1,&a
ADD R1,R0
MOV *R1,6
```

After optimization:

```
MOV R0,index
SHL R0
MOV &a[R0],6
```

7 Auxiliary Components of Compiler Phases

- The literal table
  - Usage of Literal Table

    Literal table stores constants and strings used in a program
  - Purpose of Literal Table

    The literal table is important in reducing the size of a program in memory by allowing the reuse of constants and strings

- The Symbol Table
  - Usage of Symbol Table

    Symbol table keeps information associated to identifiers: function, variable, constant and data type identifiers
  - Symbol Table with Compiler Phases
    - Scanner,parser or semantic analyzer may enter identifiers and information into the table
    - The optimization and code generation will use the information provide by the symbol table

- Error handler
  - Errors can be detected during almost every compiler phase
  - Error handler must generate meaningful error message and resume compilation after each error

1 Analysis and Synthesis

- **Analysis**
  Analyze the source program to compute its properties, include:lexical analysis,syntax analysis,and semantic analysis
- **Synthesis**
  Operations involved in producing translated code,include:code generation

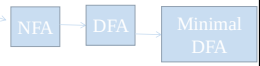
  Optimization steps may involve both analysis and synthesis

2 Front End and Back End

- **Front end**
  Operations that depend only on the source language,include:the scanner,parser,and semantic analyzer,source code optimizer
- **Back end**
  Operations that depend only on the target language, include :code generator,target code optimization

Lexical Analysis

- **Regular expression**
- **Regular grammar**



review the homework

3.2.2 Derivation and Reduction

- **Function of derivation and reduction**
  - **Context-free grammar rules determine the set of syntactically legal strings of tokens**
  - **For example:Corresponding to grammar**
    exp->exp op exp | (exp) | number
    op-> + | - | *
    (34-3)*42 is a legal string
    (34-3*42 is not a legal string
  - **Grammar rules determine the legal strings of tokens by means of derivation or reduction**

- **Leftmost and Rightmost Derivation**
  - **Leftmost derivation**
  - **A derivation in which the leftmost nonterminal is replaced at each step in the derivation**
  - **It corresponds to a preorder traversal of the parse tree**

- **Ambiguity**

A grammar G is ambiguous if there exists a string w ∈L(G) such that w has two distinct parse trees(or leftmost /rightmost derivations)

- **Categories of Top-down parsing**

1. Backtracking Parsing

- **A nonterminal has more than one productions**
- **and basing on the current input symbol, the parser can't determine which one to choose**
- **it must try different possibilities, backing up an arbitrary amount in the input if one possibility fails.**

2. Predictive parsing

Parser attempts to predict the next construction in the input string using one or more lookahead tokens

- **Two Kinds of Predictive Parsing**
  - **Recursive-descent parsing**
  - **LL(1) parsing**

Very important! review the PPT and homework!

BOTTOM- UP PARSING

1) **Right Sentential Form**
2) **Viable Prefix**
3) **Handle**

LR(0) parsing

SLR(1) parsing

Very important! review the PPT and homework!

Method of Semantic Analysis

- **Description**

  Attribute grammar is used to describe the semantic
- **Implementation**

  Syntax-directed semantics analysis

  Semantic content of a program is closely related to its syntax

Intermediate code

- **The need for intermediate code**

  Abstract syntax tree does not resemble target code, particularly in its representation of control flow constructs
- **Intermediate code**

  Representation of the syntax tree in sequential form that more closely resembles target code

- **Advantage of intermediate code**
  - **It is particularly useful when the goal of the compiler is to produce extremely efficient code**
  - **It can also be useful in making a compiler more easily retargetable**
- **Popular forms of intermediate code**

  Three-address code

intermediate code

- Reverse Polish notation
- triples
- quadruples

Bottom-up Syntax-directed translation

1 Simple assignment statement translation
2 Boolean expression translation
3 Control statement translation
4 Declaration statement translation

2. The area of the program over which the optimization applies

The categories for this classification are:

- **Local optimization**
- **Global optimization**
- **Interprocedural optimization**

Example 1:common sub expression elimination

Common sub expression is an expression that appears repeatedly in the code while it's value remains the same. Repeated evaluation can be eliminated by saving the first value for later use

(1) T_1 =4*I
(2) T_2 =addr(A)-4
(3) T_3 =T_2[T_1]
(4)T_4 =4*I ⇒ (4) T_4 :=T_1

Example 2:

Avoid storing the value of a variable or temporary that is not subsequently used

(1) I=1
(2) T_1=4
(3) T_3=T_2[T_1]
(4) T_4=T_1
(5) I=I+1
(6) T_1=T_1+4
(7) if T_1≤80 goto (3)

⇒

(2) T_1:=4
(3) T_3:=T_2[T_1]
(6) T_1:=T_1+4
(7) if T_1≤80 goto (3)

3. Costly Operations

- Reduce the cost of operations by implementing in cheaper way

  Example :

  Reduction in strength

  Replace multiplication by 2 with a shift operation

- Constant optimization

  Use information about constants to remove as many operations as possible or to precompute as many operations as possible

  - **Constant folding**

    2+3, can be computed by the compiler and replaced by the constant value 5

  - **Constant propagation**

    Determine if a variable might have a constant value for part or all of a program, and such transformations can then also apply to expressions involving that variable

2 DAG(directed acyclic graph)

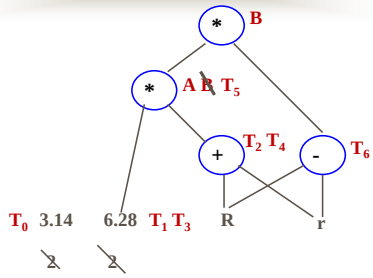
- DAG of a Basic Block

  A DAG traces the computation and reassignment of values and variables in a basic block

- Leaf nodes are values that are used in the block that come form elsewhere
- Interior nodes are operations on values
- Assignment of a new value is represented by attaching the name of the target variable or temporary to the node representing the value assigned

Example DAG of this block

(1) T_0:=3.14
(2) T_1:=2*T_0
(3) T_2:=R+r
(4) A:=T_1*T_2
(5) B:=A
(6) T_3:=2*T_0
(7) T_4:=R+r
(8) T_5:=T_3*T_4
(9) T_6:=R－r
(10) B:=T_5*T_6



- Three kinds of runtime environment
  - Fully static environment
  - Stack-based environment
  - Fully dynamic environment

- **Parameter passing mechanisms**
  - **pass by value (call by value)**
  - **pass by reference (call by reference)**
  - **pass by name (delayed evaluation)**