Two pass compiler pdf

We already know about all phases of compiler design, now the compiler passes. A compiler card refers to passing a compiler through the entire program. The success of the compiler is two types: One-pass compiler and two-pass compiler or multi-pass compiler. These are explained as follows. 1. Single Pass Compiler: If we combine or group all phases of compiler design into a single module known as single pass compiler. In the diagram above there are all 6 phases grouped into a single section, some points of a pass compiler are like: A pass/single pass compiler is this type of compiler that passes through the part of each compiler just once. The one-transit compiler is faster and smaller than the multi-pass enthroned compiler. The disadvantage of compiling a pass is that it is less effective compared to the multipass compiler. Single pass compiler is what processes the input just once, so it goes directly from the vocabulary analysis to the code generator, and then returns for the next reading. Note: Single pass compiler almost never happened, early Pascal compiler did this as an introduction. Problems with a compiler pass: We can not optimize very well because the context of expressions are limited. Since we can not back up and edit, again so grammar will be limited or simplified. Command interpreters, such as bash/sh/tcsh, can be considered as a one-pass compiler, but they also perform the entry as soon as they are processed. 2. Two Pass Compiler or Multi Pass Compiler: A two-pass/multi-pass Compiler is a type of compiler that processes the source code or abstract tree syntax of a program multiple times. In multipass Compiler we divide the phases into two passes as: First Pass: it is referred to as (a). Front end (b). Analytical part (c). Platform independent In the first pass the included phases are as Lexical analyzer, syntax analyzer, semantic analyzer, intermediate code generator is work as a front end and analytical part means that all phases analyze the high-level language and convert them three address code and the first pass is platform independent because the production of the first pass is as a three-address code that is useful for each system and the requirement is to change the code optimization phase and code generator that comes to the second pass. Second Pass: referred to as (a). Rear end (b). Composition part (c). Platform dependent On the second pass included it's like Code Optimization and the code generator is working as a back end and the composition part refers to downloading the input as three-address code and converting them to the low-level language/assembly language and the second pass depends on the platform because the final stage of a standard compiler converts the intermediate representation of the program into an executable set of instructions that depends on the system. With multi-pass Compiler we can solve these 2 basic problems: ['tü ¦pas kəm'pīl;ar] (computer science) A language processor that goes through the program will be translated twice; on the first pass controls write the sentences and construct a table of symbols, while in the second passage it actually translates the program's suggestions into a computer language. McGraw-Hill Dictionary of Scientific & Technical Terms, 6E, copyright © 2003 by McGraw-Hill, Inc. Want to thank the TFD for its existence? Let a friend know about us, add a link to this page, or visit the webmaster's page for free fun content. Link to this page: two-pass compiler 1 Compiler Construction Dr. Naveed Ejaz Lecture 2 2 2 2-pass Compiler Front End Front End Source Code IR Source Code IR Errors 3 3 Two-pass Compiler 🠳 Use an Intermediate Representation (IR) 🠳 Front End Maps Legal Source Code in IR 4 4 Two-Pass Compiler 🠳 The back end translates ir into target computer code 🠳 Admits multiple front ends & multiple passes 5 5 Two-pass compiler 🠳 The front end is O(n) or O(n log n) 🠳 The rear end is NP-Complete (NP) 2 6 6 Front End 🠳 Recognizes legal (and illegal) programs 🠳 Error reporting in a useful way 🠳 Production IR & Preliminary Storage Map 7 7 Environment Modules - Scanner 🠳 Parser Scannerparser Errors with Original Code TokensIR Errors 8 8 ScannerScanner Scanner Spertcparser Source Code Tokens IR Errors 9 9 Scanner 🠳 Character flow maps in words - basic syntax unit 🠳 Produces pairs - one word and part of speech 10 10 Scanner 🠳 Example x = x + y becomes word type token 11 11 Scanner 🠳 we call the pair a token 🠳 typical tokens: number, ID, +, -, new, whereas, if 12 12 ParserParser scannerparser kencanningsIR errors 13 13 ParserParser 🠳 Recognizes an environmentless syntax and reports errors 🠳 Guides related analysis (semantics) 🠳 Creates IR for the source program 14 14 Frameless grammar 🠳 Frameless syntax is determined by grammar G=(S,N, T,P) 🠳 S is the start symbol 🠳 N is a set of non-terminal symbols 🠳 T is a set of terminal symbols or words 🠳 P is a producer set or rewrite the rules 15 15 Without Box Grammar Grammar for expressions 1.goal→expr 2.expr → expr op term 3. | term 4.term→ number 5. | id 6.op → + 7. | - 16 16 The front end 🠳 For this CFG S = target T = { number, id, +, -} N = { target, expr, term, op} P = { 1, 2, 3, 4, 5, 6, 7} 17 17 The front end 🠳 To identify a valid sentence in a CFG, we reverse this process and create an analysis 🠳 An analysis can be represented by a tree: analysis tree or syntax tree 18 18 ParseParse Production result target 1 expr 1 expr 1 expr 1 2 expr op term 5 expr op y 7 expr - y 2 expr op term - y 4 expr op 2 - y 6 expr + 2 - y 3 term + 2 - y 5 x + 2 - y A multi-pass compiler is one compiler that processes the source code or abstract tree syntax of a program several times. This contrasts with a one-pass compilation program, which crosses the program only once. Each pass receives the result of the previous crossing as an entrance and creates an intermediate exit. In this way, the (intermediate) code improves pass by pass, until the final pass produces the final code. Multiple crossing crossing sometimes called broad compilers,[reference required] referring to the larger scope of passes: they can see the entire program being compiled, instead of just a small part of it. The wider scope thus available to these compilers allows for better code production (e.g. smaller code size, faster code) compared to the production of one pass compilers, at the cost of higher compiler time and memory consumption. In addition, some languages cannot be drawn up with a single pass as a result of their design. Typical lexical multi-pass compiler analysis This stage of a multi-pass compiler is to remove irrelevant information from the source program that syntax analysis will not be able to use or interpret. Irrelevant information could include things like comments and white space. In addition to removing irrelevant information, vocabulary analysis determines the verbal insignia of the language. This step means that the forwarding statement is generally not necessary if a multi-pass dubbing program is ever used. This phase focuses on breaking a string of characters into tokens with attributes such as genre, text, value, and possibly more. Syntax Analysis Syntax analysis is responsible for examining the rules for writing the language (often as a grammar without a frame) and for creating an intermediate representation of the language. An example of this intermediate representation could be something like an abstract syntax tree or a directed acyclic graph. Semantic analysis Semantic analysis takes the representation made from the syntax analysis and applies semantic rules to the representation to ensure that the program meets the requirements of the semantic rules of the language. For example, in the following example at the semantic analysis stage, if the language required that the conditions for whether the sentences were binary expressions the cond should be checked to make sure that it would be a valid binary expression. if(cond) { ... } another { ... } In addition to performing semantic analysis at this stage of compilation, symbol tables are often created to help create code. Code creation This final stage of a standard compiler converts the intermediate representation of the program into an executable set of instructions (often assembly). This last stage is the only stage in dubbing that depends on the machine. It can also be optimized at this stage of the collection that make the program more efficient. Other compiler passes include intermediate code production phase that takes place before code creation and the code that can be performed when the source program is written, or after an intermediate code creation phase, or after the code creation phase. Advantages of Machine Independent Multi-Pass Compilers: Since multiple passes include an modular structure and code creation disconnected from the compiler's other steps, the passes can be reused for different hardware/machines. More expressive languages: Multiple pass passes the need for forward-looking statements, allowing for the elegant implementation of mutual flashback. The main examples of languages that require forwarding declarations due to the requirement to be able to train in a single pass include C and Pascal, while Java has no forwarding statements. Reports Bornat, Richard, Understanding and Writing Compilers: A Do It Yourself Guide, Macmillan Publishing, 1979. ISBN 0-333-21732-2 Bent Thomsen, Languages and Compilers SProg og Overseatere, Department of Computer Science, Aalborg University Recovered from