TDT4205, notebook

Kim Rune Solstad

January 25, 2014

1 Introduction

1.1 Language Processors

Compiler - Program that can read a program in one language, and translate it into an equivalent program ni another language.
Interpreter - Program that directly executes operations specified in source program on inputs supplied by user.
preprocessor - Program that collects the source program befoure being further processed by the compiler.
Assembler - Program that reads assembly code and translate it to relocatable machine code.
Linker - Links relocatable machine code with relocatable object files.
Loader - Puts together executable object files into memory for execution.

1.2 Structure of a Compiler

Compiler divided into:
analysis part: Referred to as the "front end". Breaks up the source program into constituent pieces and imposes a gramatical structure on them. This structure is used to create intermediate representation of the source program. Provides informative messages if the program is ill formed. Collects information about source program and stores it in a symbol table.
synthesis part: Reffered to as the "back end". Constructs the target program from the intermediate representation and symbol table.

Lexical Analysis - First phase of a compiler. Also called tokenizing. Reads the stream of characters and groups the characters into meaningfull sequences called lexemes. A token is produced for each lexeme. A token consists of a *token-name:* abstract symbol used during syntax analysis, and a *attribute-value:* points to an entry in the symbol table.

Syntax Analysis - Second phase of the compiler. Also called parsing. Uses the first components of the tokens produced to create a tre-like intermediate representation.

Semantic Analysis - Uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. Type checking: compiler checks that each operator has matching operands.

Intermediate Code generation - Generation of an explicit low-level or machine-linke intermediate representation. This can be thought of as a program for an abstract machine. This should be easy to produce and easy to translate into the target machine.

Code optimization - attempts to improve the intermediate code so that better target code will result.

Code generation - maps the intermediate representation of the source program into the target language.

Symbol-Table management: record the variable names used in the source program and collect information about various attributes of each name. Several phases can be collected into one pass.

1.2.1 Applications of Compiler Technology

Optimizations dif computer architectures can be done through parallelism and memory hierarcies. *Memory hierarcies:* Several levels of storage with different speeds and sizes. Level closest to processor, smallest and fastest. Using registes effectivley is probably the single most important problem in optimizing a program. *Program Translation*

1. *Binary Translation:* translate binary code for one machine to that of another. Can also be used to provide backward compatibility.
2. *Hardware Synthesis:* Hardware design is usually described at the register transfer level (rtl). Hardware synthesis tools tools tanslate RTL descriptions into gates, wich are then mapped to transistors and eventually to physical layout.
3. *Database Query Interpreters:* Compiled into commands to search a database for records.
4. *Compiled Simulation:* Inputs to a simultor usually include the description of the design and specific input parameters for that particular run.

1.2.2 Software Productivity Tools

1. *type checking:* Cathces errors for example when errors are applied to the wrong type of object.
2. *Bounds checking:* Automatic range checking can prevent buffer overflows and range checks.
3. *Memory-Management Tools:* Automatic memory management obliterates all memory-management errors.

1.2.3 Programming Language Basics

Static/Dynamic: If a language uses a policy that allows the compiler o decide an issue, we say that the language uses a static policy or that the issue can be decided at compile time. On the other hand, a policy that only allows a decision to be made when we execute the program is said to be a dynamic policy or to require a decision at runtime. A language uses static scope or lexical scope if it is possible to determine the scope of a declaration by looking only at the program. Otherwise the language uses a dynamic scope.
environments and sates: The environment is a mapping from names to locations in the store. The state is a mapping from locations in store to their values. *Actual parameters:*Those used in call of a procedure. *Formal parameter:* Those used in the procedure definition.

2 Lexical Analysis

A lexical analyser reads characters from the input, and groups them into "token objects". *Removal of White Space and Comments:* Most languages allow arbitrary amounts of whitespace between tokens. Comments are lkewise ignored during parsing, so they may also be treated as whitespace. *Reading Ahead:* A general approach to reading ahead on the input, is to maintain an input buffer from wich the lexical analyzer can read and push back characters. One character read-ahead usually suffices, so a simple solution is to use a variable, say peek, to hold the next input character. The lexical analyser reads ahead only when it must. *Constants:* The job of collecting characters into integers and computing their collective numerical value is generally given to a lexical analyser, so numbers can be treated as single units during parsing and translation. *Recognizing Keywords and Identifiers:*The lexical analyser solves two problems by using a table to hold character strings

1. *Sinlge Representation:* A string table can insulate the rest of the compiler from the representation of strings, since the phases of the compiler of the compiler can work with references or pointers to the string in the table.

2. *Reserved Words:* Reserved words can be implemented by initializing the string table with the reserved strings and their tokens.

Lexical analysers can be divided into a cascade of two processes.

1. Scanning: consists of the simple processes that do not require tokenization

2. Lexical analysis: where the scanner produces the sequence of tokens

2.1 Input Buffering

Buffer pair: One pointer marks the beginning of a lexeme, another scans ahead until a pattern match is found. *Sentinels:* A special character that cannot be part of the source program. A natural choise is the character **eof**

2.2 Specification of Tokens

Regular expressions: The regular expressions are built recursivley out of smaller regular expressions. Two rules form the basis.

1. ϵ is a regular expression, and $L(\epsilon)$ is ϵ, that is, the language whose sole member is the empty string.
2. If a is a symbol in σ, then $\mathbf{a}$ is a regular expression, and $L(\mathbf{a}) = \{a\}$, that is, the language with one string, of length one, with a in its one position.

There are four parts to the induction.

1. $(r)|(s)$ denotes the language $L(r) \cup L(S)$
2. $(r)(s)$ denotes the language $L(r)L(s)$
3. $(r)^*$ denotes $(L(r))^*$
4. (r) denotes $L(r)$

2.3 Recognition of tokens

Transition Diagrams: Have a collection of nodes or circles called states. *edges:* are directed from one state of the transition diagram to another.
Recognition of Reserved Words and Identifiers: Can be done by either install teh reserved words in the symbol table initially or by creating separate transition diagrams for each keyword.

2.4 The Lexical-Analyser Generator ***Lex***

A tool that allows one to specify a lexical analyser by specifying regular expressions to describe patterns for tokens. A lex program has a structure with sections for declarations, translation rules and auxiliary functions. *Lookahead Operator:*

2.5 Finite Automata

Finite automata: acceptors for languages described by regular expressions. *Acceptor:* determines in an input string belongs to a language L. *DFA:* the transition from each state is uniquely determined by the current input character. *NFA:* There may be multiple possible choises, and some spontanious transtition without input.
The McNaughton-Yamada-Thompson algorithm can be used to construct an NFA from a regular expression.

2.6 Design of a Lexical-Analyser Generator

A *Lex* program is turned into a transition table and actions, wich are used by a finite-automaton simulator.

2.7 Optimizations of DFA-Based Pattern Matchers

functions computed from the syntax tree.

1. *nullable(n):*true iff subexpression represented by n has ϵ in its language.
2. *firstpos(n):* The set of positions in the subtree rooted at n that correspond to the first symbol of at least one string in the language of the subexpression rooted at n.
3. *lastpos(n):* The set of positions in the subtree rooted at n that correspond to the last symbol of at least one string in the language of the subexpression rooted at n.
4. *followups(n):* for a position p, is the set of positions q in the entire syntax tree such that there is some string $x = a_1a_2 \dots a_n$ in $L\big((r)\#\big)$ such that for some i, there is a way to explain the membership of x in $L\big((r)\#\big)$ by matching a_i to position p of the syntax tree and a_{i+1} to position q.

optimizations can be made by

1. *Converting a regular expression directly to a DFA*
2. *Minimizing the number of states of a DFA*

3 Syntax Analysis

LL grammar: Left to right, left recursive. *LR grammar:* Left to right, right recursive.

3.1 Context-Free Grammars

CFG consists of:

1. *terminals:* basic symbols from witch strings are formed.
2. *nonterminals:* syntactic variables that denotes sets of strings
3. *a start symbol:* nonterminal that denotes the language generated by the grammar.
4. *productions:* specifies the manner in witch the terminals and non terminals can be combined to form strings.
   (a) A nonterminal called the *head* of the production. Defines some of the strings denoted by the head.
   (b) The symbol $\rightarrow$ or ::=
   (c) A *body* consisting of zero or more terminals and nonterminals.

Ambiguity: A grammar is ambigous when it can produce more than one leftmost derivation of or more than one rightmost derivation of the same sentence. If the grammar can not be made unambigous, it is prefferrable to have disambiguating rules that throws away undesirebla parse-trees.

3.2 Writing a Grammar

*Left recursion:*A grammar is left recursive if it has a nonterminal A such that there is a derivation $A \rightarrow^{+} A\alpha$ for some string α. Top-down parsers cannot use a left-recursive grammar.
left factoring: A grammar transformation used to produce a grammar suitable for predictive or top-down parsing. Done by finding the longest prefix α common to two or more of its alternative. Add a new production with *remaining*—ϵ.