Compiling Techniques CSM201

Kristian Guillaumier
http://www.cs.um.edu.mt/~kguil
kguil@cs.um.edu.mt

Programmable Machines

- Processors are programmable in a language called **Machine Code**.
- The range of features available to this language is defined by the **Instruction Set** of the processor.
- Although the instruction set actually contains primitive, basic operations you can actually write **any program** using it.
- Each processor family has it's own instruction set. Though basic operations are common to most of them, they're incompatible with each other in almost every other respect.

A Simple Program in Machine Code (1)

- Consider the following simple statement in BASIC:

  length = 2*(side1-side2) + 4*(side3-side4)

- We'll write the equivalent machine code to execute this statement for a processor with a limited instruction set called SIMPLE.
- Variables and their values are stored in memory (RAM).
- In addition the SIMPLE processor has a single memory location (register) called the accumulator.

A Simple Program in Machine Code (2)

Memory Map

Memory Location	Variable/Constant	Value
Loc1	Length	Unknown at startup
Loc2	Side1	6
Loc3	Side2	3
Loc4	Side3	4
Loc5	Side4	2
Loc6	2	2
Loc7	4	4
Loc8	Temp	Unknown at startup

A Simple Program in Machine Code (3)

Instruction Set

Instruction	Meaning
Load <addr>	Loads a value from the memory location <addr> into the accumulator.
Sub <addr>	Subtracts a value read from <addr> from the one in the accumulator. The result is stored in the accumulator.
Add <addr>	Like above but adds.
Mul <addr>	Like above but multiplies.
Store <addr>	Stores the result from the accumulator into the memory location <addr>.

A Simple Program in Machine Code (4)

- The machine code to evaluate the expression would be:

```
Load    Loc2
Sub     Loc3
Mul     Loc6
Store   Loc8    ← we need a temp variable
Load    Loc4
Sub     Loc5
Mul     Loc7
Add     Loc8
Store   Loc1
```

Advantages and Disadvantages

- Advantages:
  - Programmers are required to have an intimate knowledge of the processor. This can lead to highly optimised code.
  - May be the only way to program the processor (like an embedded processor in a microwave oven).
- Disadvantages:
  - Programmers are required to have an intimate knowledge of the processor. Difficult to learn.
  - Development time takes longer.
  - 'Easier' to make mistakes.
  - Not portable. The program is tied down to the processor it was written for.
  - Human beings 'think' about algorithims differently than a processor does.

What is a Compiler?

- Informally a Compiler:
  - Translates a program in a language (*source language*) to another language (*target language*) usually machine code.
  - Checks for syntactical correctness.
  - Checks for semantic correctness.

Cousins of the Complier (1)

- Assemblers
  - Similar to compilers (translation/checks syntax/etc…) but the source language is Assembly Language.
- Cross-Compilers
  - The compiler program runs on a processor type, but the machine code it produces is designed to run on a different one. An example of using a cross compilers is to develop software that runs on mobile phones.
  - Cross Compilers are useful:
    - Either because the target machine doesn't have a compiler of it's own.
    - Or because it doesn't have the resources to run the compiler in the first place.

Cousins of the Complier (2)

- Interpreters
  - An interpreter translates a program into a lower level version of it, but it still cannot run directly on the processor. It depends on some runtime support. Examples of interpreted language include:
    - Command Line Interpreters (BASH, command.com)
    - Batch Files
    - VBScript, JavaScript
  - Execution is slower since the translation occurs each time the program is executed.
  - Interpreters are easier to write.

A Deeper Look into Compilers

- To keep things manageable the process of compilation is separated into 3 distinct (though connected) phases:
  - Lexical Analysis
  - Syntax and Semantic Analysis
  - Code Generation

Lexical Analysis

- Lexical Analysers are also called Scanners.
- Recall that a program is made up of many small entities:
  - Keywords: IF, THEN, ELSE, …
  - Identifiers: counter, my_var, openfile
  - Numbers
  - Symbols: +, /, >, >=
- Put Simply, the scanners job is to:
  - Open the source file,
  - Recognise the entities and represent them as tokens,
  - Remove Comments,
  - Produce error reports.

Example

```
for counter = 1 to 20 print "hello world" next
```

Tokens

for	Keyword	
counter	Identifier	Name = Counter
=	Operator	Equals
1	Constant	Value = 1
to	Keyword	
20	Constant	Value = 30
print	Keyword	
“hello world”	String	Value = “hello world”
next	keyword	

Syntax Analysis

- The syntax analyser is also known as the **Parser**.
- For all the compiler is concerned, the sequence of tokens produced by the scanner is just a random sequence of symbols. It is the job of the syntax analyser to ensure that these symbols are structured correctly according to the definition of the language. For example:
  - Every BEGIN must match an END in Pascal.
  - Every statement must end in a semi-colon.

Semantic Analysis

- Even though the structure of the language is correct, the MEANING of the statements may be invalid according to the semantics of the language.

  my_var = my_var + 1

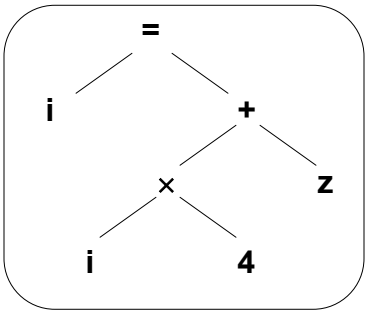
  is correct in terms of syntax, but is my_var is declared as a string, the arithmetic addition of a number to a string isn't really correct.

The Symbol Table and Parse Tree

- The output produced by the Syntax and Semantic Analyser is the:
  - Symbol Table:
    - Stores information about identifiers and functions, such as their types, sizes, names, number of arguments etc…
  - Parse Tree:
    - Stores the structure of the program.

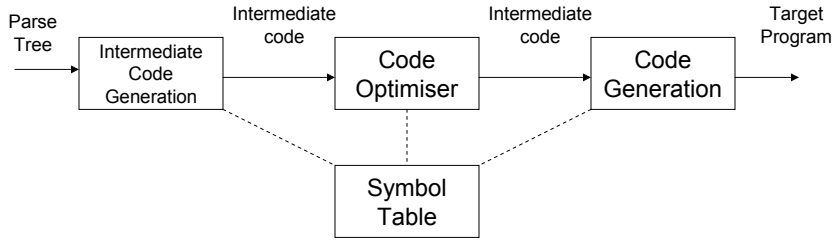
Example Parse Tree

- i = (I * 4) + z



Code Generation

- After all the preceding phases have been completed successfully without errors, the compiler will proceed to build the target code from the data structures previously constructed.
- In many cases Code Generation of further split:



Intermediate Code Generation

- The code generator starts off by generating an intermediate form of code representation before actually building the target code.
- The main difference between intermediate code and the actual target code is that certain details such as the exact memory locations are omitted.
- A common representation format for intermediate code is the **Three-Address Code**.

Three-Address Code

```
temp1 = 60
temp2 = id2 + temp1
id1 = temp2
```

Operator	ARG1	ARG2	Result
=	60		temp1
+	id2	temp1	temp2
=	temp2		id1

Code Optimisation

- This phase attempts to rearrange the code to obtain a smaller or faster running version.

```
temp1 = 60
temp2 = id2 + temp1
id1 = temp2
```

Equivalent to:

```
id1 = id2 + 60
```


Front and Back Ends (1)

- Commonly compiler phases are split into two different categories:
  - The front end: this stage is concerned with the phases related to the source language and are independent of the target. This part usually consists of the lexical analyser, syntax analysis, symbol table creation, semantic analysis and intermediate code generation.
  - The back end: consists of the stages dependent on the target machine. This usually consists of the code generation and certain parts of the optimiser.

Front and Back Ends (2)

- Splitting the compiler into front and backends has the following advantages:
  - The backend can be modularly changed to compile the same source language for a different platform.
  - Compilers for different source languages usually produce standard intermediate code and may reuse the same backend.

Language Specification

- Programming languages must be specified and properly described before attempting to write a compiler for them.
- The specification is written in a **meta-language**.
- Meta-Languages need to be unambiguous and we rely on **Formal Languages** to assist.

Formal Languages Primer (1)

- In order to specify a formal language rigorously we need to introduce some concepts:
- A **Symbol** or **Token** is an atomic (indivisible) entity usually a character, digit or keyword.
- An **Alphabet**, denoted by Σ, is the finite, non-empty set of symbols.
- A **String** over the alphabet is a sequence $a_1a_2\ldots a_n$ of symbols from Σ.
- The symbol ε denotes the empty string.
- $\varepsilon a = a\varepsilon = a$

Formal Languages Primer (2)

- The set of all strings over the alphabet Σ, including the empty string ε, is denoted by the **Kleene Closure** - Σ^*.
- The set of all strings over Σ, whose length is at least 1 (i.e. does not contain ε), is denoted by the **Positive Closure** - Σ^+.
- A **Language** L over the alphabet Σ is a subset of Σ^*.

Regular Expressions

- Many languages (though not all) may be described using a notation called **Regular Expressions**.
- Regular expressions specify strings in a language by using symbols from it's alphabet and a few special meta-symbols:
  - **Concatenation:** when we wish to concatenate symbols or string we write them next to each other of use the . (dot) meta-symbol for extra clarity.
  - **Alternation:** when there is a choice between to symbols α and β, they are separated by the | (bar) symbol.
  - **Repetition:** a symbol α followed by a * (star) indicates that there are zero or more repetitions of α.
  - **Grouping:** a group of symbols may be grouped by surrounding them by the meta-symbols (and) – parenthesis.

An Example

- Consider the expression:

```
1 ( 1 | 0 )* 0
```

- This expression represents all the strings that start with a 1, end in a 0 and have an unlimited (possibly empty) number of 1's and 0's in between.

```
{10, 100, 110, 1000, …}
```

Notes on Regular Expressions

- Precedence from highest to lowest: Parenthesis → Repetition → Concatenation → Alternation

  ab* ≠ (ab)*

- If the meta-symbols are part of the alphabet, they should be enclosed in quotes. For example, comments in Pascal would be:

  "(" "*" c* "*" ")"

  where $c \in \Sigma$

- Another convention normally used is that of the + repetition instead of the *. It has the same meaning as the Positive Closure. Basically it's a shortcut for writing aa*

Algebraic Properties of Regular Expressions

A | B = B | A	Commutativity for alternation
A | (B | C) = (A | B) | C	Associativity for alternation
A | A = A	Absorption of alternation
A . (B.C) = (A.B).C	Associativity for concatenation
A (B | C) = AB | AC	Left distributivity
(A | B) C = AC | BC	Right distributivity
A ε = ε A = A	Identity for concatenation
A* A* = A*	Absorption for closure

Regular Expressions

- Description of Identifiers

```
(_|A| … |Z|a| … |z).(_|A| … |Z|a| … |z|0| … |9)*
```

- Description of Integers

```
(0 | 1 | 2 | … | 9)+
```


Grammars

- Formally a **Grammar** is a quadruple {N,T,P,S} where:
  - N is the finite set of non-terminal symbols,
  - T is the finite set of terminal symbols (Σ),
  - P is the finite set of production (or grammar) rules,
  - S is the starting, goal or sentence symbol.
- A sentence is a string entirely composed of terminal symbols.

Example

- Consider the following language:

ε
ab
aabb
aaabbb
aaaabbbb
...

- The grammar for the above language is:

({S}, {a,b}, P, S)
where P is:
S → ε
S → aSb

Another Example

- Consider the following rules for a context sensitive grammar:

1) S → aSBC
2) S → aBC
3) CB → BC
4) aB → ab
5) bB → bb
6) bC → bc
7) cC → cc

- The following is a *derivation* from S to a valid string:

S
aSBC (by rule 1)
aaBCBC (by rule 2)
aaBBCC (by rule 3)
aabBCC (by rule 4)
aabbCC (by rule 5)
aabbcC (by rule 6)
aabbcc (by rule 7)

Types of Grammars

- The complexity and structures of the rules in a grammar determines what types of languages we can describe and recognise using it. These different "Grammar Types" are categorised by the **Chomsky Hierarchy**:
  - Type 0 – Unrestricted Grammars
  - Type 1 – Context Sensitive Grammars
  - Type 2 – Context Free Grammars
  - Type 3 – Regular Grammars

Unrestricted Grammars

- Productions take the form:

$$A \rightarrow \alpha$$

where,

- A and α are arbitrary symbols in the vocabulary $\mathbf{N} \cup \mathbf{T}$.

Context Sensitive Grammars

- Productions take the form:

$$\alpha A \beta \rightarrow \alpha \gamma \beta$$

where,

- $A \in N$
- $\gamma \neq \varepsilon$
- $\alpha, \beta, \gamma \in (N \cup T)^*$
- May also include the rule $S \rightarrow \varepsilon$

Context Free Grammars

- Productions take the form:

$$A \rightarrow \alpha$$

where,

- A is a single non-terminal symbol ($A \in N$),
- α is a, possibly empty, string of terminals and/or non-terminals.

Regular Grammars

- Productions take the form:

(i)		(ii)
$A \rightarrow \alpha$	-or-	$A \rightarrow \alpha$
$A \rightarrow \alpha B$		$A \rightarrow B\alpha$

The difference between the two is that one is right recursive (since B can be equal to A) and the other is left recursive. Regular grammars must either be one or the other, but never both (otherwise this would be a type 2 grammar)

Backus-Naur-Form (BNF)

Symbol	Meaning
::=	**'is defined as'**
|	**Or**
< >	**Angles brackets surround category symbols – These symbols are called *Non-Terminals***
[ ]	**Optional items are surrounded by square brackets**
{ }	**Zero or more repetitions of an item are surrounded by curly brackets**
" "	**Exact symbols in the language are enclosed in quotes – These symbols are called *Terminals*. Sometimes the quotes are omitted.**
;	**End of line**
ε	**Empty or Nothing**

BNF By Example (1)

Consider the identifier:
my_variable

In plain English:

Identifiers consist of any sequence alpha-numeric characters and the underscore symbols. However an identifier cannot start with a digit.

Formally in BNF:

```
<ident> ::= <alpha> | "_" {<alpha>|<digit>|"_"};

<alpha> ::= "a"|"b"|…|"z"|"A"|"B"|…|"Z";

<digit> ::= "0"|"1"|…|"9";
```


BNF by Example (2)

Problem:

Construct a BNF specification for simple expressions limited to integer numbers and identifiers. The operators allowed in this type of statement are + and –.

For Example:

```
3

counter

counter + 1

counter + (1 – y)

3 + 4
```

BNF by Example (2½)

```
<expr> ::= <factor> | <factor> <op> <expr>;

<factor> ::= <integer> | <ident> | "(" <expr> ")";

<op> ::= "+" | "-";

<integer> ::= <digit> {<digit>};
```


Extensions to BNF (EBNF)

- In order to improve the readability and conciseness of descriptions in BNF several extensions have been proposed to the notation. BNF with these extensions is called EBNF.
- *Kleene Cross*: a sequence of *one or more* items of a class are:

```
<unsigned-int> ::= <digit>+
```

- *Kleene Star*: a sequence of *zero or more* items of a class are:

```
<ident> ::= <letter><alphanumeric>*
```

- Braces are used for *grouping* instead of the usual 'zero or more' interpretation.

```
<ident> ::=  <letter>{<letter>|<digit>}*
```

An Example in EBNF

- Consider BNF for variable declarations in Basic:

```
<var-decl> ::= "dim" <var-decl-list>

<var-decl-list> ::=
     <var-decl-item> {ε | "," <var-decl-list>}

<var-decl-item> ::= <ident> "as" <var-type>
```

- In EBNF could would be written as:

```
<var-decl> ::= "dim" <var-decl-list>

<var-decl-list> ::=
     <var-decl-item> {"," <var-decl-item>}*

<var-decl-item> ::= <ident> "as" <var-type>
```