CS 241 - Foundations of Sequential Programming

Kevin James

Fall 2013

Definition

Sequential programs are procedurally driven programs which only use a single process at a time. They are single-threaded and do not run processes concurrently. For this course, we are not interested in what those programs do, but simply in how they do it.

These programs can be written in machine language, assembly, higher-level languages... whatever. Since each is an abstraction of a lower level to which we can build a compiler, all of them will give us the same final result.

Data Representation

A computer sees a string such as "10000011" as a sequence of bits, without regards to any sort of encoding. This could be binary, hexadecimal, decimal, ascii... pretty much everything, but the computer can not know which.

Hexadecimal Representation

Hexadecimal is represented with the characters "01234567899ABCDEF". We tend to write them in the form $0x2A$, which is equal to 42 in decimal.

Negative Numbers

With n bits, we can represent 2^n numbers in binary. To represent negative numbers we can use the **sign and magnitude** (use the first bit as an indicator of sign and the remaining bits as magnitude) method, though this is subject to some weirdness. We will end up with both a positive and a negative zero and will need specialized hardware for both addition and subtraction.

A better method is to use **two's complement** (uses the congruences of numbers to represent their negatives). We can convert a number into two's complement form by the following method:

-6 can be converted by taking the absolute value in binary (0110), then flipping all of its bits (1001) and adding 1 (1010). Obviously this method can be used backwards: subtract 1, flip the bits, and add a negative number.

Alternatively, subtract 2^n if and only if the first bit is non-zero (to go *from* two's complement *to* decimal).

Parts of a Processor

A processor is comprised of a **control unit**, which controls the execution of the program and keeps track of the next instruction to execute. It can interpret strings of binary as command instructions and controls the signals to and from any external devices. It contains a timing device for synchronised operations. The processor also contains an **arithmetic logic unit**, which is responsible for arithmetic and logic operations.

The **program counter (PC)** is a 32-bit register which contains the address of the next instruction to be executed. The **Instruction Register (IR)** contains the current instruction address.

We also have **General Purpose Registers**thirty-two zero-indexed registers which are 32-bits in size though they are somewhat slow.

Range of Registers

Our registers range from \$0 − \$31. Register \$0 always contains the value 0 and register \$31 always contains the value of the return address. Register \$30 contains the address of the top of the stack.

How a Program is Executed

The OS invokes the "loader" to load the program into memory starting at some given memory address, virtually always zero. The loader places the starting address in the PC: "PC $\rightarrow 0x00$".

We then fetch, decode, and execute each instruction in order. In effect:

```
while(1) {
  fetch        // IR <- MEM[PC]
  increment    // PC <- PC + 4
  decode
  execute
}
```

MIPS

MIPS commands are lines of binary which designate the function to be executed and the registers involved. For example, in the ADD command we have

000000	*sssss*	*ttttt*	*ddddd*	00000	100000
function code	\$s	\$t	\$d	padding	opcode

which will give us $\$d = \$s + \$t$. The subtract command is similar, but uses the opcode 100010.

Another useful command is the jump command jr \$s. This command basically executes

```
PC = $s
```

and thus jumps to that address. Note that this is necessary for exiting our program, since the address for returning our program is initially stored in register \$31.

LIS is another useful command; the LIS command will run

```
$d = MEM[PC]
PC = PC + 4
```

Which can be used to assign a register to any value of your choice.

Assembly Language

There is a one-to-one correspondance between each assembly and machine instruction. The assembly can be converted directly into machine language with an **assembler**.

An assembler will read in a file and scan through it, recognizing each line to be a discrete instruction. It will remove comments and divide the instructions into **words** and **tokens**. This first step is referred to as **scanning** (taking the sequence of characters and outputting a sequence of tokens). This is also refered to as lexial analysis or tokenization.

There are multiple types of tokens, for example: opcode (ID), register (REG), comma. These tokens will also be associated with what we refer to as a **lexene** (the string which was recognized as a given token). For example, the lexene of a REG token may be `$29`.

Loading

A **loader** is a program that places a file into RAM. For N instructions, we do:

```
for(int i = 0; i < N; i++) {
  MEM[i] = file[i];
}
```

Note that we can't necessarily assume that we begin at memory address zero. That is where **MERL** (MIPS Executable Relocatable Locatable code) comes into play. We can add additional information to our file at either the beginning (`beq $0, $0, n; ...; begin:`) or the end (`jr $31; ...`). Often we use both: the beginning of our code contains the address of the relocation table at the end of our code.

In effect, a **relocator** will read in the first n bytes of code, follow those to memory addresses, and add α to them (after the code has already been compiled by assuming an initial `pc` of zero). For example, if a program beginning with `example: .word example` is meant to be run beginning at memory address $0x8$, this first line would be first compiled as 0000 0000 then relocated as 0000 1000.

Relocating

A relocator uses the following algorithm (note that the relocation table follows the sample format `.word 1; .word x; .word 1; .word y...`

```
i = alpha + MEM[alpha + 8]
end = alpha + MEM[alpha + 4]

while i < end:
  if MEM[i] == 1:
    MEM[alpha + MEM[i + 4]] = MEM[alpha + MEM[i + 4]] + alpha
  i += 8
```

or, in words

1. `i` is the address of the start of the relocation table
2. `end` is the address of the end of the relocation table
3. check if `MEM[i]` has value 1
4. get the original address of a location to be relocated (`MEM[i + 4]`) and overwrite it with the location after the offset (`MEM[i + 4] + alpha`).

Linking

If we want to combine multiple assembly files, we can not simply concatenate them. Especially if labels are used in some file but defined in another, we must use a **linker**. A linker takes two or more MERL files, edits the header, concatenates the MIPS code, and edits the relocation tables (now refered to as the **footer**) to contain both sets of relocation tables as well as some additional linker information.

The additional information is an **external symbol reference**, which interacts with **external symbol definitions** in other files to define labels cross-files. For both of these sections, we encode the label name as ASCII then represent either the relocatable location or the relocatable value respectively.

We use the assembler directive `.import print` to tell our assembler to expect `print` to be a label defined in an external file. If we do not use this line, but refer to the `print` label in our program, the assembler should error. If we include this line, but do not actually include the external file, the assembler should error anyway: this `.import` directive is only a courtesy line for our assembler.

We use the `.export` directive to do the opposite: this will inform our assmelber that the label following the directive will be used in an external file, and that the assembler should thus add this label to our MERL footer (i.e. as an external symbol definition).

We can see, then, that the linker provides the matches between ESD's and ESR's which allows for the definitions in all files to be valid.

Formal Languages

A **formal language** is a mathematically precise way of specifying what consists of a language. It consists of an alphabet (single tokens which must be taken on their own - not always a-z, sometimes

the alphabet may be { cat, dog, horse... }), words (finite sequences of the alphabet), and the special word ε which signifies an empty or zero-length word. Together, this is a language, which is a potentially infinite set of words.

For example, we may have a finite language which defines "all 8-bit numbers in binary" or an infinite one which defines "all prime numbers in decimal".

A **recognition algorithm** is a decision algorithm which takes in the specification of a language and some input string and determines whether this string is in the defined language or not.

Language Classes

The classes of language are as follows, from easy to recognize to difficult:

1. Finite
2. Regular
3. Context-free
4. Recursive
5. Undecidable

Note that all decidable classes can be recognized with a recognition algorithm. Also, higher-order recognition algorithms can be used on lower classes of languages.

Finite Languages

A **finite language** is made up of a finite set of words, for example: { car, cat, cow }. It is specified by enumerating the words in the language, and can be recognized by comparing the input to all words in the specification.

For the above example, we would read in our input character by character. The first must be a c, the next an a or o, and the final a r, t, or w (depending on what the second character is), and we fail as soon as we find input which is not possible.

Regular Languages

A regular language is one for which you can specify a **finite automata** or **regular expression**.

Deterministic Finite Automata

A **DFA** is a 5-tuple $(\Sigma, Q, q_0, A, \delta)$ defined by:

- Σ: a finite set of symbols
- Q: a finite set of states
- $q_0 \in Q$: an initial / starting state
- $A \subseteq Q$: any non-zero amount of accepting / final states

- δ: the transition function such that $Q \times \Sigma \to Q$

For example, the DFA represented by $(\{a, b\}, \{0, 1\}, 0, 1, \{(0, a, 1), (0, b, 0), (1, a, 0), (1, b, 1)\}$ accepts any input with an odd number of a's. Any b's it receives simply "loop back" to whatever state it is currently in.

The standard psuedo-code for a DFA is as follows:

```
state <- q0
for i 1 to n
  state <- delta(state, token(i))
end for
if state in A
  return true
else
  return false
```

If we are in a state such that our next input token is not a valid transition, we must fail instantly.

Non-deterministic Finite Automata

An NFA is a finite automata such that there exists at least one state where a given symbol would cause a transition to two or more states simultaneously. There is always an equivalent DFA for any possible NFA, but using an NFA can sometimes simplify the problem.

An NFA is defined much the same as a DFA, with perhaps a few more delta transitions. It can be approximated with:

```
states <- {q0}
for i 1 to n
  states <- U delta(states, token(i))
end for
if states ^ A != emptyset
  return true
else
  return false
```

Epsilon Transitions

Epsilon transitions give us ε**-NFA**'s, which are similar to NFAs, but can transition without consuming any input. Basically, an ε-NFA allows you to move from a given state to any number of others spontaneously.

We define the ε-closure of any state as the set of states which can be reached from that state using zero or more ε transitions.

Regular Expressions

Regular expressions allow us to represent regular languages as a series of symbols using pre-defined control characters. For example, we have "n,x" which tells us the preceeding character must appear

between n and x times or "—" which tells us to expect a single token from either side of this symbol (but not both).

For example, to match an HTML hex-based color code we might use `#?([a-f0-9]{6}|[a-f0-9]{3})`.

Context-Free Grammar

A regular language can not be used to represent nested structures such as a potentially infinite number of parenthesis, as is possible in most object-oriented languages. Instead, we must use a **context-free language**, which can be specified with a context-free grammar as follows: `LHS` $\rightarrow$ `RHS`.

The LHS of every CFG rule must be a non-terminal character and the RHS must be any number of terminals and / or non-terminals.

For example, the ruleset

```
expr -> expr op expr
op -> +|-|*|/
expr -> term
term -> a|b|c|d...
```

can be used to **derive** the input string "a + b". Beginning with an `expr` we can use the following derivation:

```
S expr
expr -> expr op expr
  expr -> term
    term -> a
  op -> +
  expr -> term
    term -> b
```

If we include a rule such as `expr` $\rightarrow$ `( expr )`, we find ourselves with a CFG which can accept a potentially infinte level of nested brackets.

We formally specify CFGs as follows: each is a 4-tuple (N, T, P, S) where we have

- N is a finite set of non-terminals
- T is a finite set of terminals
- P is a finite set of production rules
- S is a unique non-terminal with which we start

Canonicity

If we always expand the left-most terminal first, we will find ourselves with a left-most / **left canonical** derivation. Similarly, expanding the right-most terminals first gives us a **right canonical** derivation. For any given input string, these two derivations are each unique.

Ambiguity

If we can capture the same input stream with more than a single derivation (i.e. there exist multiple ways of creating a derivation such that we match the same string), a grammar is said to be **ambiguous**.

WaterLanguage for Programming with Pointers

WLPP is a subset of C++. It has the following restrictions:

- Two types: `int` and `int*`.
- One function (i.e. no procedures):
  - `int wain(int a, int b)` or `int wain(int* a, intb)`
- The last statement must be a `return` (then some close braces).
- `if` statements must have a matching `else` block, even if it is empty.
- Uses custom `println` to print to the screen.
- Operators: plus, minus, times, divide, `*`, `&`, `new`, `delete`.

Compilers

A compiler must first scan the input into a series of tokens. The tokens must then be run through a parser to determine their meaning and a context-senstive analyzer to ensure correctness. At this point, we may optionally optimize the generated code before outputting it.

Scanning

This step of the compilation process involves reading in the input and determining its membership in the language defined by the compiler. The scanner then (i.e. if it finds valid input) outputs a series of tokens; these tokens will be fed into the parser to determine program validity.

This is the step in the process where we may catch syntax errors such as invalid characters, missing whitespace, et cetera.

Maximal Munch Algorithm

The **maximal munch algorithm** (for regular languages) reads in as many symbols as it can until it gets "stuck" in a state such that it can not move to a different state. If this state is accepting, it returns that state as a token; otherwise, it backtracks until it is in an accepting state, outputs that, then begins from the start of its DFA with the characters which it "spit back out" as well as the remaining input string.

Parsing

The parsing step of the compilation process takes in the tokens generated by the scanner and ensures logical validity. The parser then (i.e. if the input is valid) outputs a parse tree to be used in context-sensitive analysis.

At this level of the compiler, we are guaranteed to have only tokens which are valid symbols in our language; the parser simply ensures that those tokens are in a logically sound order. It is at this level that we may catch errors such as missing tokens (especially semi-colons and braces...), malformed expressions, and the like.

Top-Down Parsing

A top-down parser requires an input string and an **augmented unambiguous grammar** and outputs a derivation / parse tree. It should be of $O(n)$ complexity.

When we are comparing an input string to our grammar, we follow the following rules:

1. Look at the current α (e.g. for $S => B\ C$, the α is $B\ C$) starting at the left-most symbol.
2. If that symbol is terminal, match it to the corresponding symbol in the input string and repeat step 1 with the next symbol. If we can't match the terminal symbol, throw an error.
3. Otherwise (if the symbol is non-terminal), pick a rule to expand the non-terminal. Replace the LHS of the rule with the RHS and repeat step 1 with the next symbol.
    - When chosing a rule to apply, we must pick a rule which matches the input via look-ahead. For example, if we have rules $S => ab, S => ad$ and input string ab, then for non-terminal S we should choose rule $S => ab$ for this step, as the replacement will then match our input.

For the sake of efficiency, we tend to put the input string through a stack: as we match each character, we can pop them off and thus never have to match them again.

The most common parsing format is LLk, where we have **L**eft-to-right **L**eft-most derivation with **k** symbols of look-ahead.

Bottom-Up Parsing

Bottom-up parsers take in the same output as top-down parsers, but perform the opposite set of logical equations. The input string is anaylzed to give us the derivation rules which were used to create it, instead of performing the reverse analysis.

There are a few common operations which bottom-up parsers use:

- **Shifting** consumes one symbol from the input by pushing it onto the stack.
- **Reducing** takes the RHS of a rule and pops it off of the stack, the pushes its LHS onto the stack.

We always try to reduce before shifting. Note that by following these operations, we will find ourselves with a backwards, right-most derivation.

One thing worth noting is that though we "always try to reduce before shifting", this step of the process is more complicated than it sounds. Sometimes, we find ourselves in situations where we must accept more input though we already have something reducable, and then reduce several times in a row later.

In this case, using a DFA for our "reducability" state can be a good idea: we may craft one in which we only reduce when the look-ahead character definitely can not put us into a larger reducable state.

Take, for example, the partial input string "$a+b-c$". If we have rules $A \to a+term, term \to b, term \to c, term \to b-c$, then we *can* reduce after reading in only three symbols. However, if we do this, we will find our string is unparseable. Better yet is to read in this token, see the $-$ ahead of it, and continue reading in until we find ourselves with $term \to a-b$. Only at this point should we reduce our input: $a+b-c \to a+term \to A$.

The generally accepted method for creating a bottom-up parser is to keep a symbol stack and a state stack. The symbol stack is pushed onto whenever we shift, and is popped from n times and pushed onto once whenever we reduce. The symbol stack acts as a pointer to our position in a DFA which keeps track of when we should reduce.

By developing a DFA which keeps track of this, we can simply query the top of our state stack to determine if we are in a reducable state. If we are, then we can pop off the same number of items which we pop off of the symbol stack, and push on whichever state corresponds to the item we will be pushing onto the symbol stack. This ensures we never have to read through our input twice to determine reducability.

For the following CFG, we have developed a DFA such that the four derivation rules (below) are valid at states 4, 8, 5, and 6 (in that order). The full DFA should be easy enough to derive.

```
S -> BOF E EOF
E -> E + T
E -> T
T -> id
```

The following table shows the state of our various stacks and variables as we parse through the input string `BOF id + id + id EOF`

Symbol Stack	State Stack	Read Input	Remaining Input	Action
null	1	null	BOF id + id + id EOF	shift BOF
BOF	1 2	BOF	id + id + id EOF	shift id
BOF id	1 2 6	BOF id	+ id + id EOF	reduce T → id
BOF T	1 2 5	BOF id	+ id + id EOF	reduce E → T
BOF E	1 2 3	BOF id	+ id + id EOF	shift +
BOF E +	1 2 3 7	BOF id +	id + id EOF	shift id
BOF E + id	1 2 3 7 6	BOF id + id	+ id EOF	reduce T → id
BOF E + T	1 2 3 7 8	BOF id + id	+ id EOF	reduce E → E + T
BOF E	1 2 3	BOF id + id	+ id EOF	shift +
BOF E +	1 2 3 7	BOF id + id +	id EOF	shift id
BOF E + id	1 2 3 7 6	BOF id + id + id	EOF	reduce T → id
BOF E + T	1 2 3 7 8	BOF id + id + id	EOF	reduce E → E + T
BOF E	1 2 3	BOF id + id + id	EOF	shift EOF
BOF E EOF	1 2 3 4	BOF id + id + id EOF	null	reduce S → BOF E EOF
S	null	BOF id + id + id EOF	null	null

As you can see, we've finished with only our start symbol on the symbol stack, which is the ultimate goal of the validity-checking portion of a bottom-up parser. The input string is most certainly a logically valid input, thus we can output our parse tree and send control to the context-sensitive analyzer.

Context-Sensitive Analysis

Once code has gone through the parser, it is guaranteed to be syntactically correct. However, there are still other properties which must be checked.

Linear-bounded Automata are technically possible, but tend to take more effort to design than they are worth. A typical CSA tool will utilize tree-traversal methods instead.

The CSA is responsible for ensuring variables have been used properly and for ensuring type correctness.

- **Variable Correctness** A CSA must ensure that variables are declared before being used and are not declared multiple times
- **Type correctness** In addition to variables being of the correct type, expressions must be as well. The CSA must ensure that any comparisons compare identical types, that any multiplication is done only with `int`s, that the `delete[]` function only accepts pointers, that variables are defined with the proper type (e.g. `int* pointer = 42;` is not valid)...

Code Generation

Once our code has reached the code generator, we know that it is syntactically correct. Any errors from this point on will be runtime errors and as such can not be caught by a compiler.

The code generator is responsible for transforming the parse tree into valid Assembly code. It tends to do this in pieces, by finding a declaration, for example, and transforming this into the assembly equivalent.

Optimizations

Optimizations can be done at any point in the compiler's process, and can typically have a greater effect when we begin with earlier levels of the compiler. For example, if we decide to use an **abstract syntax tree** instead of the standard parse tree (i.e. as output from our parser), then we may find it simpler to perform our optimizations later on.

The bulk of our optimizations tend to occur either immediately before or during the code generation phase.

Constant Folding

If our compiler were to come across an expression such as `x = 1 + 2`, we could simplify this in the compiler to `x = 3`, thus reducing the number of instructions required to implement this.

Constant Propogation

At any point where we are guaranteed to know the value of a variable, we can simply substitute that value into any equations it is a member of. By combining this with constant folding, we can greatly reduce the amount of generated code.

For example, if we have the code block

```
int a = 30;
int b = 9 - a/5;
int c;

c = b * 4;
if(c > 10) {
  c = 2;
}
return c * 2;
```

we can greatly simplify this using constant folding and constant propogation alone. Applying both, we find ourselves with

```
int a = 30;
int b = 3;
int c;

c = 12;
if(true) {
  c = 2;
}
return c * 4;
```

Dead Code Elimination

This type of optimization is also quite effective when paired with the previous two optimizations. For the above example, we can see that removing the variables `a` and `b` is possible, as is removing the `if` block (i.e. forcing the contained code to execute without bothering to do a test.

We can thus reduce the block of code to

```
int c;

c = 12;
c = 2;
return c * 2;
```

and by applying these concepts further, simply to `return 4;`

Register Allocation

When dealing with assembly, we have registers and a stack in which we can store variables (be they temporary or not). So if we find ourselves commonly generated code such as

```
lis $1
.word 1
lw $3, -4($30)    // x
add $3, $3, $1
sw $3, -4($30)    // x + 1
```

we may find it is simpler to keep `$14` (for example) as a constant reference to the number 1.

Strength Reduction

Some instructions may be represented in multiple ways, in which case it may be beneficial to chose the fastest of them.

For example, we may represent `2 * x` as either

```
lis $2
.word 2
mult $3, $3
mflo $3
```

or more simply as `add $3, $3, $3`.

Common Sub-Expressions

We may find ourselves with an expression such as `x = 2*y % 2*y`, especially after other optimizations have modified the code. In this case (where we have the same sub-expression present on both sides of an operator, we may simplify this without even knowing the value of that sub-expression.

- $x + x = 2x$
- $x \times x = x^2$
- $x - x = 0$
- $x \ \% \ x = 0$
- $x \ / \ x = 1$

Memory Management

Memory management may be either implicit (i.e. garbage collection) or explicit (i.e. with `alloc` and `free`, in `C++`).

Scope vs Extent

The **scope** of a variable is the part of the program under which the variable's name is visible. The scope is determineed statically, thus the compiler can be responsible for dealing with that variable (e.g. placing it on the stack when it is declared and removing it when we have left that scope.

The **extent** is the part of the program where the contents of the variable's memory is live. Extent is a property of the data, whereas scope is a property of the variable. For stack-based allocation, the scope and the extent are identical. However, memory management comes into play when you want the extent to outlast the scope. This data, then, should not be put on the stack, but on the heap (or we should keep the stack offset as a constant).