Contents

1 Foundation of Sequential Programs

A sequential program runs its instructions one at a time.

- Machine Language (A1)
- Assembler
- Assembly Language (A2)
- Compiler (A6- A10)
- C/C++/Racket

1.1 Data Representation

$$67 = 64 + 2 + 1 = 2^6 + 2^1 + 2^0 = 1000011$$

Decimal, Binary, Octal, Hexadecimal Representations

Unsigned binary: $0 - 2^n - 1$

Sign and magnitude:

- Two representations of 0
- Additional and subtraction require separate hardware

Two's complement:

- Use congruence to represent negative numbers
- Represent negative number as $N - b$
- Positive numbers are unchanged
- Flip the bits and add one for negative numbers
- Signed binary number: Look at first bit

Aside: Let a, b be two positive numbers. $(a - b = a + (N - b)) \bmod N$ where $N = 2^n$

$N - b$ is easy to do because $(N - 1 - b) + 1$ and flipping bits is fast

Converting two's complement to decimal

- Treat number as unsigned number and convert to decimal
- If first bit represents positive number, done
- If first bit represents negative number, subtract N

1.1.1 Comparisons

Signed and unsigned need separate comparisons. Static typing helps select the comparison type.

1.2 Machine Language

Machine Language: Set of machine instructions. Understood by a particular machine/processor

Machine Instruction: Sequence of bits having exactly one meaning

1.2.1 MIPS Processor

- Control Unit: Controls flow of data, timing, and executing the program
- ALU: Arithmetic Logic Unit: Math operations
- Memory: 32bit (word)
- Register: Small amount of very fast memory (32 bits each), (MAR, MDR)
- Program Counter: Special register that always holds address of the next instruction to execute

- Instruction Register: Holds current instruction (32 bits)
- 32 General purpose registers (Use $ to represent registers)
- Register 0 always contains the value 0

Processor communicates with memory through data bus

To run a program, the loader loads the program into memory starting at address 0.

Fetch-decode-execute algorithm:

- Hardcoded as a circuit in the control unit
- Sets program counter to 0
- **Program counter**: Contains the address of the next instruction

```
loop {
    IR <-- MEM[PC]
    PC <-- PC + 4
    decode
    execute
}
```

Note: Explicitly need to jump out of the fetch-decode-execute cycle to exit the program

`$31` contains a special instruction that is similar to the return statement in C++.

1.3 MIPS

Clear value of a register `$add $4 $0 $0`

Negate value of a register `$sub $4 $0 $4`

Copy value to another register `add $3 $0 $1`

Terminate program: `jr $31`

`xxd -b file.bin` shows binary values

Example: Add 42 and 52 and store the result in `$3`

Solution:

Load immediate: `lis $d`

`d <- MEM[PC], PC <- PC + 4`

```
lis $5
.word 42
lis $7
.word 52
add $3 $5 $7
jr $31
```

Assembly Language: Textual representation of machine language: Less tedious, less error prone, abstraction over machine language.

`cs241.binasm` converts assembly to binary

Example: Compute the absolute value of `$1` and store the result in `$1`

Set Less Than: `slt $d $s $t`

Branch on Equal: `beq $s $t i` - If `$s == $t`, `pc = pc + i * 4`

Branch on Not Equal: `bne $s $t i`

Solution:

```
slt $2, $1, $0
beq $2, $0, 1
sub $1, $0, $1
jr $31
```

Alternate Implementation:

```
slt $2, $1, $0
bne $2, $0, 1
jr $31
sub $1, $0, $1
jr $31
```

Unconditional Jump: `beq $0 $0 5`

Example: Calculate $13 + 12 + 11 + \cdots + 1$ Store result in `$3`

Solution:

```
lis $2
.word 13
add $3, $0, $0
lis $1
.word -1
add $3 $3 $2
add $2 $1 $2
bne $2 $0 -3
jr $31
```

`.word loop` is the address of the instruction

Load Word: `lw $t i($s)`

- Loads a word from memory.
- i is 16 bits signed 2's complement'
- `$t <- MEM[$s + i]`

Store Word: `sw $t i($s)`

- Store a word from a register into memory
- `MEM[$s + i] = $t`

1.3.0.1 Writing Standard Output

Magic address: `0xffff000c`

If something is stored at this address, least significant byte is outputted to `stdout`

```
lis $1
.word 0xffff000c
lis $2
.word 67
sw $2, 0($1)
```

Outputs ASCII value for 67 which is `C`

Multiplication: `mult` and `multu`

- `mult $s, $t`
- There is no target register
- Result may be bigger than 32 bits
- `hi:lo` register is used
- `hi` stores most significant bits and `lo` stores least significant bits

Division: `div` and `divu`

- `lo` is quotient `$s / $t`
- `hi` is remainder `$s % $t`

Accessing `hi` and `low`

- Use `mfhi` and `mflo`
- `mfhi $4` moves content of high register into register 4

Example:

- `$1` contains address of a 32 bit array of integers, denote this `arr`
- `$2` contains length of array
- Goal: Read from `arr[3]` and store in `$3`

Suppose `$1` is 40. `arr[3]` would be at 52 since $40 + (3 \times 4) = 52$

```
lis $4
.word 4
lis $3
.word 3
mult $3, $4
mflo $5 ; Has (4 x 3)
add $5, $1, $5
lw $3 0($5)
jr $31
```

Alternate implementation: `lw $1 12($5)`

1.4 Testing Arrays

```
mips.array program.mips
```

1.5 Procedures in Assembly

A **label** is associated with the address at that instruction by the assembler

Procedure: Label followed by instructions

Issues:

- Prevent procedure from overwriting other procedure's register values
- Calling and returning from a procedure

Issue 1: - `$3` contains critical data - Call procedure

```
proc:
    overwrite $3
    return
```

Value in `$3` is lost

Solution 1: Restricting procedures to a subset of registers

- Does not scale
- Disallows recursion

Solution 2: Use Memory (RAM)

- MIPS loader initializes `$30` to just past memory assigned to the program
- When a procedure is called, the procedure stores the previous register values in memory and updates `$30` bookmark
- Before returning, the new procedure restores the old register values. It updates `$30` again
- Using memory as a stack and treating `$30` as the stack pointer

```
---------------------------------------
Code: MEM[0]

End of Code
---------------------------------------
empty space



Procedures memory use this area

----------------------------------------
$30: End of allocated memory for program
----------------------------------------
Other program's memory
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
----------------------------------------
```

Example: `push $1`

- Take value in $1 and push it to the top of the stack.

```
sw $1, -4($30)
lw $1
.word 4
sub $30, $30, $1
```

Example: `pop $5`

```
lw $5
.word 4
add $30, $30, $4
lw $5, -4($30)
```

Issue 2:

- How to return to address that called procedure?
- `beq` and `jr` will not work because no reference back
- Need to remember the program counter when calling a procedure

```
lis $8
.word proc
jr $8
```

This code can jump to `proc`, but how do you return back?

Jump and Link Register: `jalr $s`

- `$31 = PC, PC = $s`
- At the end of the procedure, `jr $31` takes you back to original location

Problem: This overwrites previous `$31`. How do you terminate program?

```
lis $8
.word proc
push $31
jalr $8
pop $31
jr $31 ; Returns properly :)
proc:
    do stuff
    jr $31
```

Note: Push and pop are not functions in MIPS. Must implement it with our implementation

Convention: Use registers unless a procedure requires 30+ variables.

Procedures document argument expectations

1.5.0.1 Recursion

Works out of the box

Example:

```
sw $31 , -4($30)
lis $4
.word -4
add $30, $30, $4
lis $4
.word proc
jalr $4
lis 4
.word 4
add $40, $40, $4
lw $31, $-4(30)
...
jr $31
proc:
    ; Push registers that may be overwritten
    ; Procedure code
    ; pop registers (restores registers)
    jr $31
```

1.6 Assembler

1.6.1 Analysis

1.6.1.1 Input

- Scanning/tokenization (Reading and splitting input)
- Convert a sequence of characters into a sequence of tokens

```
struct token  {
    Kind kind; //ID, REG, COMMA (enum)
    char * lexome; // Characters from input
    long long value; // constant
}

vector<Token> line;
```

1.6.1.1.1 Parsing

- Make sense of the sequence of tokens

```
if (line[0].Kind = 11) &&
   (line[0].lexeme == "add") &&
   (line[1].Kind == REG) &&
   (line[2].Kind == COMMA) &&
   (line[3].Kind == REG) {
    ...
   } else {
    // syntax error
    // output ERROR to stderr
   }
```

- Check for valid syntax
  - Length of line
  - Range of registers
  - Range for branches (16 bit signed)

1.6.1.2 Labels in Assembly

- Branch (beq): `(offset = address of label - current PC)/4`
- `.word` label
- Assembler should keep track of virtual PC

Symbol Table: Store the address of each new label into a map

If we see `.word label`, lookup and output the address from the symbol table.

In `beq`, lookup label in table, and use the formula to calculate offset.

- There is no declaration before use for labels
- Assembler will need at least 2 passes.
  - Pass 1: Scanner and most syntax checking, creates symbol table
  - Pass 2: Remaining syntax checks, range checks, and generate output

Example:

```
main: lis $2
    .word 13
    lis $1
    .word -1
    add $3, $6, $0
loop:
begin: top: add  $3, $3, $2
    add $2, $2, $1
    bne $2, $0, top
end: jr $31
```

Pass 1: Generate Symbol Table

```
main -> 0
loop -> 20
top-> 20
begin ->20
end ->32
```

Loader: Read program from file on disk, and load it into ram.

1.6.1.3 OS 1.0

```
repeat:
    Choose program
    loader(p)
    jalr $0 // Assuming we load program at address 0
    beq $0, $0, repeat
```

1.6.1.4 Loader 1.0

```
for (int i = 0; i < words; ++i) {
    MEM[4i] = file[i]
}
```

Problem: Cannot always assume that we can load the program at address 0

1.6.1.5 OS 2.0

```
repeat:
    p <- Choose program
    $3<-loader(p) // Starting address of where the program should be loaded
    jalr $3
    beq $0, $0, repeat
```

1.6.1.6 Loader 2.0

Assume a is the starting address of enough continuous memory to hold the program and associated stack.

```
for (int i = 0; i < words; ++i) {
    MEM[a + 4i]= file[i]
}
```

Problem: Assembler assumes that the program is going to be loaded at address 0. Any line that contained a `.word <label>` is incorrect if the program does not start at address 0.

Solution: Add a to each line containing `.word <label>`. Assembler must provide information on locations of `.word <label>` to the loader.

Assembler will produce object code. (Machine code with additional information)

1.7 MERL: MIPS Executable, Relocatable, Linkable

Assembler provides more information about the program:

- `beq $0, $0, 2`. This skips the next two lines and goes to the first line of the program
- End of file address
- End of code (Relocation entries)

`0x00000001` tells loader that the next line contains the address of a line that must be relocated

1.8 Loader

Output MERL object code

- Header: 3 words
- Code
- Footer/table

1.8.0.1 Tools

`cs241.relasm` generates MERL output.

`cs241.merl` relocate a MERL file to a specific address and outputs a non-NON MERL file

`mips.twoints` and `mips.array` have an optional command line argument to provide starting address

```
java cs241.relasm < inputasm > out.merl
java cs241.merl 0x12345678 < out.merl > out.mips
java mips.twoints out.mips 0x12345678
```

1.8.0.2 MERL Output without `relasm`

- Add new labels for each occurrence of `.word label`
- Add label for end of code
- Add relocation entries
- Add label for end of module
- Add header:

```
beq $0, $0, 2
.word endmodule
.word endcode

.word A
.word B
endcode:
.word 1
.word reloc1
.word 1
.word reloc2
endofmodule:
```

1.8.0.3 Loader 3.0

```
----------------
memory
---------------
start of program
beq $0, $0, 2
addr endmodule
addr endcode
---------------
code
endcode:
---------------
footer
endmodule:
---------------
```

Need the address if the symbol table. This is at `MEM[a+8]`.

The actual address is `MEM[a+8] + a`

```
start <- MEM[a+8] + a
end <- MEM[a+4] + a
while (start < end) {
    if (MEM[start == 1])
        MEM[MEM[start + 4] + a] += a
        start += 8
    }
}
```

1.9 Linkers

Problem: How to resolve labels that are defined in a different file?

Solution 1: Concatenate all files and run assembler on concatenated files

Limitation:

- Need to assemble entire project every time.
- Inefficient
- How to deal with relocations? Only one file can be loaded at address 0
- Concatenating MERL files produces two tables

Solution 2: Use a tool that understands how to combine MERL files

Limitation: Since we will assemble first and then link, the assembler has to tell the linker of any unresolved labels.

- Assembler will output (in the MERL footer) an entry to indicate that an external symbol reference is still unresolved
- We do not want the assembler to add an ESR (External Symbol Reference) entry for every label that's not found.
- To use an external symbol, programmer must import

```
.import <label> ; Allows reference <label> externally
```

If assembler sees `.word proc` but no `.import proc`, the assembler will generate an error

```
.word 0x11 ; next addr is an ESR (External Symbol Reference)
.word <addr> ; address where word is imported
.word <length of label>
.word <ascii value 1 of label>
...
.word <ascii value of nth letter in label>
```

If we want labels to be visible to other files, we must use `.export <label>`

For each exported label, an External Symbol Definition (ESD) is created.

```
.word 0x05 ; next addr is an ESD (External Symbol Definition)
.word <addr> ; address where word is defined
.word <length of label>
.word <ascii value 1 of label>
...
.word <ascii value of nth letter in label>
```

When linking the files together, it will look like

```
header
file1 code
file2 code
```

`file2` code will be offset by `m1code - file2.header`

- Match all ESR from `file1` with ESD from `file2`
- Once matched, update the value of the `.word <label>` using the address from the ESD and remove the ESR.
- Replace the ESR with the relocation entry

1.10 Formal Languages

An implementation (assembler/compiler) is useful to check what is allowed/not allowed in a language.

Limitation: Who guarantees that the implementation is correct?

A **formal language** is a mathematically precise way of saying what is/isn't allowed by a language.

Noam Chomsky used mathematics (set theory) to create a hierarchy of languages called the Chomsky Hierarchy.

Definitions:

Alphabet: (denoted $\sum$) a finite set of symbols. (Example: $\{0, 1\}$ is the binary alphabet)

Symbols can be multiple characters. Example $\sum = \{jr, add, \$1\}$

Word: An ordered finite sequence of symbols (Example `jr $31` is a word in the MIPS alphabet)

There is a word ϵ, the word with 0 symbols.

Language: A possibly infinite set of words.

Example: $\sum = \{0, 1\}$ and $L =$`{ all binary numbers of length 8}`. Therefore $|L| = 2^8$

The empty language, denoted ϵ is the language with 0 words.

Specification: Precisely define what is/isn't valid

Recognition: A decision algorithm that takes a specification of an input word and answers whether the word is in the language.

Note that there are languages which do not have a recognition algorithm.

Interpretation/Translation: Converting input to a desired output.

Assembler/Compiler must do both recognition (analysis) and translation (synthesis)

How hard is it to do recognition? It depends

Classify languages based on how difficult their recognition algorithm is. (Easy to hard)

- Finite Languages
- Regular Languages
- Context Free Languages
- Context Sensitive Languages
- Recursive Language
- Undecidable Language

1.11 Finite Languages

Language which has a finite set of words.

Specification: List all the words in the language.

Recognition: Check if a given word is in the set of words that define that language

Example: $L = \{cat, co, car\}$ To check if $w \in L$, start with checking whether first letter is c. If not, can immediately terminate and say no. Trie seems to work really well here

Move through transitions, changing state every time. A state is something like "start", "seen c", "seen ca", "seen cat", etc.

Example: $L =$ {all MIPS keywords}

1.12 Regular Languages

Building Blocks:

- Finite languages
- Union
- Concatenation
- Repetition

Union: $L_1 \cup L_2 = \{x : x \in L_1, x \in L_2\}$

Concatenation: $L_1 L_2 = \{xy : x \in L_1, y \in L_2\}$

$L_1 = \{dog, cat\}, L_2 = \{fish, \epsilon\}$

$L_1 L_2 = \{dogfish, dog, catfish, cat\}$

Repetition: $L^* = \{\epsilon\} \cup \{xy : x \in L^*, y \in L\}$

A regular language can be specified using a DFA, NFA, ϵ-NFA, or RE.

1.13 Deterministic Finite Automaton

Regular languages (DFA) are used in writing scanners.

Formal Definition: A DFA is a 5-tuple $(\sum, Q, q_0, A, \delta)$

- $\sum$: Alphabet
- Q: Set of states
- q_0: Starting state
- A: Set of accept states
- δ: $f : (x \in Q, y \in \sum) \rightarrow z \in Q$. Applies transition to a state

1.14 Recognition

Input $w = a_1 a_2 \ldots a_n, a_i \in \sum$

$L(DFA) \rightarrow$ yes/no is $w \in L(DFA)$

```
state = q
for i in 1 to n:
    state = sigma(state, a_i)
end for

return (state in A)
```

Note: A word is accepted by the DFA if the recognition algorithm returns true, the given the word leads the DFA into an accept state.

The language specified by a DFA is the set of all words accepted by the DFA.

A language is regular if there exists a DFA for it.

Example: Words with the same number of a's and b's in any order (this is not a regular language because states will be infinite)

1.15 DFA with actions/ Finite Transducers

$$\sum = \{0, 1\}$$

$L = \{$ a binary number with no leading zeros $\}$

In a finite transducer, we can associate an action with a transition.

Assume n is a global variable.

At a branch in the DFA, modify the global variable.

1.15.0.1 DFA Accepting all BIN/DEC/HEX

```
(0) -> 0 -> (0)
(0) -> 1-9 -> () -> loop 0-9
(0) -> 0 () -> x -> () -> 0-9/A-F -> (()) -> loop -> 0-9/A-F
```

This is not a DFA because the expression is ambiguous and is not deterministic. This is a Non deterministic finite automaton.

NFAs can also be used to represent regular languages. A word is accepted by an NFA if a path stops at an accept state.

1.16 Non Deterministic Finite Automaton

- $\sum$: Alphabet
- Q: Set of states
- q_0: Starting state
- A: Set of accept states
- $\delta : (Q \times \sum) \to P(Q)$ where $P(Q)$ is the power set of Q. (all possible subsets of Q)

$L(NFA) = \sum, Q, q_0, A, \delta$

Yes/no, is $w \in L(NFA)$

```
states = {q_0}
for i in 1 to n:
    states = \Union_{s_j \in states} sigma(s_j, a_i)
end for
return states intersect A > 0
```

Note: DFAs are easier to implement while NFAs are easier to create.

Epsilon NFAs have an epsilon as the first transition to multiple states. The transition does not consume any symbol.

Example:

$$\sum = \{a, b, c\}, L_1 = \{cab\}, L_2 = \{\text{ even number of a's}\}$$

```
() -> c -> () -> a -> () -> b (())
() -> (()) -> loop -> <- b,c -> a -> () -> loop -> bc
```

To merge the two NFAs, we can use epsilon NFA, and use epsilon as the starting transition for both NFAs.

```
() -> epsilon -> () -> c -> () -> a -> () -> b (())
() -> epsilon  -> () -> (()) -> loop -> aa -> a -> () -> loop -> bc
```

Epsilon closure of a state S is the set of all possible states that are reachable from the states in S using 0 or more epsilon transitions.

Input $w = a_1a_2 \cdots a_n, a_i \in \sum$

```
states = epsilon_closure(q_0)

for i in 1 to n:
    state = epsilon_closure(\Union_{j \in states}) \sigma(s_j, a_ijj)
end for
return states intersect A > 0
```

1.17 Regular Expression

- $epsilon$: Empty word
- $a, a \in \sum$: The word matched by a is in the alphabet
- R_1R_2: A word matched by R_1 followed by a word matched by R_2 (Concatenation)
- $R_1|R_2$: Either word R_1 or the word R_2 (Union)

- R^*: 0 or more occurrences of words matched by R

Convention: $a|bc^* \equiv a|(b(c^*))$

Example: $L = \{cab, car, card\}$

Regular Expression: $cab \mid car \mid card$ or $ca(b|r(\epsilon|d))$

Example: $\sum = \{a, b\}$, $L = \{$all strings containing aa$\}$

$(a|b)^*aa(a|b)^*$ is an ambiguous regex for this.

Kleene's Theorem: A language is regular if

- there exists a regular expression for it
- there exists an ϵ NFA for it
- there exists an NFA for it
- there exists a DFA for it.

$$RE \iff \epsilon - NFA \iff NFA \iff DFA$$

Proof:

$$RE \implies \epsilon - NFA$$

- $\emptyset$ is just the start state
- ϵ is the valid start state
- a is a transition with a
- $R_1|R_2$ can be achieved by using an epsilon transition to connect to start state of R_1 and R_2
- R_1R_2 is connecting each finish state in R_1 to start state in R_2. R_1's finish state will not be valid states anymore
- R^* is creating a new state as start and accepting state, connect it to start state with epsilon, and loop back to start state for every valid end state (with epsilon transition).

$$\epsilon - NFA \implies NFA$$

- For every chain of ϵ transitions followed by a transition on a symbol a, add the transition a from the beginning of the chain to the state a transitions to.
- If the target state of an ϵ transition is an accept state, make the source state also an accepting state.
- Remove all ϵ transitions and any unreachable states.

$$NFA \implies DFA$$

This is called subset construction.

- An NFA can transition to multiple states. Construct a state that is a subset of all possible states given a particular transition. Then there on each transition, we can only be in one state at a time.
- Any accepting state in the original NFA that is a subset of a state will have the new state as an accepting state.

1.18 Recognition in Scanning

- Input DFA, NFA
- Input: $w = a_1, \cdots a_n, a_i \in \sum$
- Output: yes/no is $w \in L$

1.18.1 Scanning

Converting a sequence of characters into a sequence of tokens. (Often referred to as tokenization)

- Input: String s
- Input: DFA
- Output: $w_1 w_2 \cdots w_n = s$

$L = \{$tokens for programming language$\}$

Suppose this language is regular.

$M_L =$ DFA for recognizing a single word from L.

Machine can recognize any amount of words by adding an ϵ transition from every accept state to the start state. This accepts LL^*

During ϵ transition, output the token (as an action) and we can generate a sequence of tokens.

1.18.1.1 Problem

$L = \{aaa, aa\}$

This isn't deterministic. $aaaaa$ can be formed with $\{aaa, aa\}$ or $\{aa, aaa\}$

$s = aaaa$

Either $\{aa, aa\}$ valid, or $\{aaa\}$ and then the input is invalid.

1.19 Maximal Munch Algorithm

- Greedy algorithm
- Always deterministic
- Might sometimes not give tokenization (even if one exists)

Always takes the maximum token length at every transition.

- Input: $aaaaa$, Output: $\{aaa, aa\}$
- Input: $aaaa$, Output: $\{aaa\}$ and then the input is invalid.

1.19.0.1 Algorithm

- Start with the DFA that recognizes a single word
- Run input program through this DFA
- Acceptance:
  - If at accepting state, good
  - Otherwise backtrack to last seen accepting state
  - If no such state exists, output error
- Output a token
- reset current state to start state

Example:

```
int i = 1
int j = 1
println(i+++j) // i++ + j
println(i+++++j) // i++ ++ + j is invalid
println(i+++ ++j) // i++ + j++ is valid

vector<vector<int >> // the >> gets interpreted as bit shift and cannot compile in C++03
vector<vector<int > > //  Solution: Put space

y/*z  // This will not compile as /* is the beginning of a comment */
y/ *z // Solution is to put a space
```

1.19.0.2 Semantics

- Input: string s
- Input: L(DFA)
- Output: $s = w_1, w_2, \cdots, w_n, w_i \in L(DFA)$ and w_i is the longest prefix from $w_i, w_{i+1}, \cdots w_n$

Worst case scenario:

$L = abc|(abc)^*d$

Input: $abcabcabcabcabcabc$

Correct separation: $\{abc, abc, abc, abc, abc, abc\}$

However, maximal munch will match $(abc)^*d$ until the very end, and then back-track.

Worst case: $O(n^2)$

1.20 Simplified Maximal Munch

When stuck at an non-accepting state, just output error.

WLP4 tokens from the same class must be separated by whitespace

```
i = 0
state = q_0
loop
    if (\delta(state, s_i) is defined):
        state = \delta(state, s_i)
        i++
    else:
        if (state is not Accepting):
            error quit
        output token
        state = q_0
        if EOF:
            quit
end loop
```

1.21 Parsing

Parts of a compiler

Input -> Scanner -> Sequence of tokens (Lexically correct) -> Parser -> Parse tree (Syntax correct) -> Semantic analysis / Context sensitive analysis

-> parse tree + symbol table + types, etc -> semantically valid -> Code generator -> MIPS assembly

Parsing: Figuring out structure of the program.

Example: Structured Mathematical Operations. Will need infinite number of states in DFA to represent an arbitrary number of balanced parenthesis. (Using a regular language)

Context Free Languages:

- Specified using content free grammar
- expr -> id
- expr -> expr op expr
- id -> a | b | c
- op -> + | - | * | /

Context Free Grammar is a set of rewrite rules. Replaces repetition with recursion

Start with the start symbol (expr), and replace it with the corresponding RHS of a rule

$a + b$

- expr $\implies$ expr op expr
- $\implies$ id op expr
- $\implies$ id + expr
- $\implies$ id + id
- $\implies$ a + id
- $\implies$ a + b

1.21.0.1 Adding Parentheses

Add new rule:

expr $\implies$ (expr)

Example:

- expr $\implies$ (expr)
- $\implies$ ((expr op expr))
- $\implies$ (((expr) op expr))
- $\implies$ (((expr op expr) op expr))

1.21.0.2 Goal

- Specify programming language syntax using CFG.
- Start at start symbol of CFG and try to derive the input (output of the scanner)
- If we cannot find one, parsing error
- Valid -> create parse tree

Formally, a Context Free Gramar is a 4 tuple

- N: Finite set of non terminals
- T finite set of terminals
- S: Unique start symbol ($S \in N$)

- P: Finite set of production rules
- V: $N \cup T$

$P \subseteq N \times V^*$

$A \to \alpha$ where $A \in N$ and α is a sequence of terminals and non-terminals

- Non terminal can appear on the LHS or RHS of a rule
- Terminal can only appear on the RHS
- There must be at least one rule for each non-terminal

1.21.0.3 Conventions

- $a, b, c, d \in T$ (single terminal)
- $A, B, C, S \in N$ (single non-terminal)
- $W, X, Y, Z \in V = N \cup T$ (single symbol)
- $w, x, y, z \in T^*$ (string of terminals)
- $\alpha, \beta, \gamma \in V^*$ (string of terminals/non-terminals)

1.21.0.4 Definitions

Directly Derive: A single application of a rule

$\alpha A \gamma \implies \alpha\beta\gamma$ if $A \implies \beta$ is a rule.

Derive: we say $X_1 \implies {}^*X_n$ if $X_1 \implies X_2 \implies \cdots \implies X_{n-1} \implies X_n$

$\alpha \implies {}^*\beta$ if $\alpha = \beta$ or $\alpha \implies \gamma$ and $\gamma \implies {}^*\beta$

Derivation: A sequence $\alpha_0, \alpha_1, \cdots, \alpha_n$ such that

- a_0 is the start symbol
- a_n is a string of just terminals
- $\alpha_i \implies \alpha_{i+1}$

Show $expr \implies {}^*a + b - c$

- expr $\implies$ expr op expr
- $\implies$ expr op expr op expr

- $\implies$ id op expr op expr
- $\implies$ a op expr op expr
- $\implies$ a + expr op expr
- $\implies$ a + id op expr
- $\implies$ a + b op expr
- $\implies$ a + b - expr
- $\implies$ a + b - id
- $\implies$ a + b - c

Definition: The language defined by the CFG G is the set of strings of terminals that can be derived starting at the start symbol S.

$$L(G) = \{x \in T^* | S \implies {}^* x\}$$

The language is context free if it can e specified using a CFG.

It is undecidable whether two CFGs specify the same language

Recognition: yes/no is $s \in L(G)$

Parsing: Prove that $s \in L(G)$ by providing a derivation

Example:

- $N = \{A, B, C\}$
- $T = \{a, b, c, d, e, f, g, h\}$
- $S = A$ (Start symbol)
- $A \implies BgC$
- $B \implies ab$
- $B \implies cd$
- $C \implies h$
- $C \implies ef$

$L = \{abgh, cdgb, abgef, cdgef\}$

Parse $abgef$

- $A \implies BgC \implies abgC \implies abgef$
- $A \implies BgC \implies Bgef \implies abgef$

We have two derivations for the same string

1.21.0.5 Parse Tree

- Root - start symbol
- Leaf Nodes: Terminals
- Node A can have children WXY if $A \implies WXY \in P$

Properties:

1. A derivation uniquely defines a parse tree

    - expr $\implies$ expr op expr
    - op $\implies$ + | - | * | /
    - expr $\implies$ id
    - $T = \{+, -, *, /\}$

    Input string: id - id * id

    - exp $\implies$ expr op expr
    - $\implies$ expr op expr op expr
    - $\implies$ id op expr op expr
    - $\implies$ id - expr op expr
    - $\implies$ id - id op expr
    - $\implies$ id - id * expr
    - $\implies$ id - id * id

2. A parse tree can have multiple derivations (but derivation uniquely defines parse tree)

    - exp $\implies$ expr op expr
    - $\implies$ expr * expr
    - $\implies$ expr * id
    - $\implies$ expr op expr * id
    - $\implies$ id op expr * id
    - $\implies$ id - expr * id
    - $\implies$ id - id * id

3. An input string can have multiple derivations (similar to parse tree example).

    - Order of operations doesn't exist. Therefore multiple derivations exist.

1.22 Leftmost Derivation (Left Canonical Derivation)

- Always expand the leftmost non-terminal.
- $\underbrace{xA\gamma}_{\alpha i} \implies \underbrace{x\beta\gamma}_{a_{i+1}}$
- for $A \implies \beta$

1.22.0.1 Rightmost Derivation

- Always expand rightmost non-terminal.
- $\gamma Ax \implies \gamma\beta x$
- for $A \implies \beta$

Property: From a given derivation style, a parse tree has a unique derivation.

Problem: The two derivations previously for $id - id * id$ are both leftmost derivations.

Ambiguous Grammar: A grammar is ambiguous if there exists a string in the language for which there exist multiple parse trees, even when using a fixed derivation style.

1.22.0.2 Unambiguous Grammar

No proof to show unambiguity. Undecideable problem

- expr -> expr op term | term
- term -> id | (expr)
- op -> + | - | * | /

Note: Parenthesis added just to show order of operations and evaluation in parse tree

- expr $\implies$ expr op term
- $\implies$ (expr op term) op term
- $\implies$ (id op term) op term
- $\implies$ (id - term) op term

- $\implies$ (id - id) op term
- $\implies$ (id - id) * term
- $\implies$ (id - id) * id

What if we want order of operations? We want multiplication and division to be deeper than addition and subtraction in the parse tree.

Give * and / precedence over + and -.

- asdf -> expr pm term | term
- pm -> + | -
- term -> term md factor | factor
- md -> * | /
- factor -> id | (expr)

Note: Parenthesis added just to show order of operations and evaluation in parse tree

- expr $\implies$ expr pm term
- $\implies$ (term) pm term
- $\implies$ factor pm term
- $\implies$ id pm term
- $\implies$ id pm (term md factor)
- $\implies$ id pm (factor md factor)
- $\implies$ id pm (id md factor)
- $\implies$ id pm (id * factor)
- $\implies$ id pm (id * id)

Example:

```
int x = 0;
int y = 8;
if (x < 5)
    if (y > 10)
        print "A"
    else
        print "B";
```

A warning will appear "dangling else". Without putting braces, the expression is ambiguous. Unknown whether else belongs to first or second if statement. Note that the compiler doesn't look at whitespace and indentation.

2 ——————- Midterm Stops Here —————————-

2.1 Parsing

- Input: Unambiguous grammar and tokens
- Action: Parse in $O(n)$
- Output: Derivation

Top down parsing: Start with root symbol and work down

Bottom up parsing: Start with derivation and build upwards

Parsing algorithms "like" having only one rule for the start symbol

Augmenting a Grammar:

$G' = \{N \cup \{S'\}, T \cup \{\vdash, \dashv\}, S', P \cup \{S' \implies \vdash S \dashv\}\}$

S' is the start symbol

where $\vdash$ is BOF and $\dashv$ is EOF

Algorithm:

- When we see terminals, match with input
- Find a rule for the first leftmost non-terminal ($A \to B$)
- Replace LHS of rule with RHS
- $S' \implies \vdash S \dashv$
- $S \implies AyB$
- $A \implies ab$
- $A \implies cd$

- $B \implies 3$
- $B \implies wx$

Input: $\vdash aywx \dashv$

$S' \implies \vdash S \dashv \implies \vdash AyB \dashv$

Now there are two rules for A to match. Use the next yet to be matched symbol (look-ahead) to predict

`Predict[A][a]` = $\{A \implies ab\}$

$\implies \vdash AyB \dashv \implies \vdash abyB \dashv$

`Predict[B][w]` = $\{B \implies wx\}$

$\implies \vdash abywx \dashv$

Efficiency: $\alpha_i = xA\beta$

Why keep matching x to the input when it has already been done once?

Solution: Use a stack to keep track of yet to be matched portion of α_i

2.1.0.1 Stack Diagram

Refer to online notes for stack diagram

Possible errors:

- Top of stack does not match first unmatched symbol fro input -> **Parsing error**
- `Predict[A][a]`: 0 rules $\implies$ Parsing error
- `Predict[A][a]`: multiple rules $\implies$ Language not LL(1)

LL(k) Algorithm:

- Left to right scan
- Leftmost derivation
- Use k next symbols to lookahead and determine what rule to apply.

Typically languages use LL(1) or bottom up parsing as prediction table would be too big.

2.2 Bottom up Parsing

`Predict[A][a]` = $\{A \to B | B \implies {}^*a\gamma\}$

`First(A)` = $\{a | A \implies {}^*a\beta\}$

`First(A)` has start terminals for all strings that can be derived from A.

Example:

- $S \implies AB$
- $A \implies \epsilon$
- $B \implies b$

Input: b

Predict(A,a) = $\{A \to \beta | a \in First(\beta)\}$ OR $Nullable(\beta)$ and $a \in Follow(A)\}$

$Nullable(\beta)$ is true if $B \implies {}^*\epsilon$

$Follow(A) = \{a \,|\, S' \implies {}^*xAa\gamma\}$

Example:

- expr -> id
- expr -> expr op expr
- op -> +

Predict[expr][id]

- Using expr->id, id ∈ First(id)
- Using expr -> expr op expr, id ∈ First(expr op expr) ∈ First(expr) ∈ First(id)

Thus Predict[expr][id] $= \{id, expr\, op\, expr\}$

There are two rules for the prediction table so grammar is not LL(1)

Example:

- S -> AB
- A -> ϵ
- A -> a

- B -> a

No nullable non-terminal should have a terminal "A" in its first and follow set.

`Predict[A][a]` = $\{A \implies a, A \implies \epsilon\}$

There are multiple rules in the prediction table so the grammar is not LL(1). Should not be more than one way to derive ϵ for a nullable symbol.

Example:

- S -> AB
- A -> ϵ
- A -> B
- B -> ϵ
- B -> a
- $S \implies AB \implies B \implies a$
- $S \implies AB \implies BB \implies aB \implies a$

Example:

- S -> AB
- A -> ϵ
- A -> a
- B -> ϵ
- B -> a

Multiple derivation rules

- $S \implies AB \implies B \implies a$
- $S \implies AB \implies aB \implies a$

The same non-terminal should not generate the same first terminal through two different rules.

- E -> E + T | T
- T -> T * F | F
- F -> a | b | c

Even though grammar is unambiguous, it is still not LL1 because

- E -> T -> F -> a
- E -> E + T -> T + T -> F + T -> a + T

The first terminal is the same for both. Left recursive grammars are never LL(1)

Try making the grammar right recursive

- E -> T + E | T
- T -> F * T | F
- F -> a | b | c
- $E \implies T + E \implies F + E \implies a + E$
- $E \implies T \implies a$

Same problem still exists.

Note: Productions for a non-terminal should not have a common prefix on the RHS.

- E -> TE′
- E′ -> + E | ϵ
- T -> FT′
- T′ -> * T | ϵ
- F -> a | b | c

2.3 Bottom up Parsing

$w \leftarrow \alpha_{n-1} \leftarrow \alpha_{n-2} \cdots \leftarrow S'$

- Read input string from left to right
- Replace the RHS of a rule with LHS
- $S' \implies \vdash S \dashv$
- $S \implies AyB$
- $A \implies ab$
- $A \implies cd$
- $B \implies 3$
- $B \implies wx$

Will use a stack to store $\alpha_i s$

Algorithm can perfor the following actions

- Shift: Push the next symbol from the input on the stack
- Reduce: Pop the RHS of a rule from the stack and push the LHS

Consumed Input	Remaining Input	Stack	Action
abywx		shift	
a	bywx	a	shift
ab	ywx	ba	shift
ab	ywx	A	reduct A->ab
aby	wx	yA	shift
abyw	x	wyA	shift
abywx		xwyA	shift
abywx		ByA	reduce B->wx
abywx		S	reduce S->AyB
abywx		/ S	shift
abywx		S′	reduct S′ -> S

Bottom up parsing gives a reversed rightmost derivation.

RIBS: Always Reduce It Before Shifting

Example

- $S' \implies \vdash E \dashv$
- $E \implies E + T$
- $E \implies T$
- $T \implies id$

$E + T$ needs to be on TOS.

Notation to represent how much of a RHS is on the stack

Item: An item is a rule with a bookmark somewhere between the symbols on the RHS of a rule.

- $E \implies \cdot E + T$ (Fresh item, none of the RHS is on the stack)

After pushing onto the stack,

- $E \implies E \cdot +T$
- $E \implies E + \cdot T$
- $E \implies E + T \cdot$

This is an reducible item

To model this, create a DFA for all items for all rules in the grammar.

Since running the contents of the stack again and again through the DFA is inefficient, always keep track of the state we are in.

Option 1: Interleave symbol and states on the stack

Option 2: Maintain a symbol stack plus a state stack

This algorithm is LR(0). Left to right scan, rightmost derivation, 0 lookahead

Parsing error occurs when there is no transition defined in the DFA. for the current state and symbol that was pushed. (Refer to bottom up parsing handout)

When is a grammar not LR(0)?

Shift reduce conflict:

- $A \implies \alpha \cdot AB$
- $B \implies \gamma \cdot$

Reduce reduce conflict:

- $A \implies \gamma \cdot$
- $B \implies \beta \cdot$

For a grammar to be $LR(0)$, the $LR(0)$ DFA should not have shift reduce or reduce reduce conflicts.

- $S \implies \vdash E \dashv$
- $E \implies T + E$
- $E \implies T$
- $T \implies id$

At this point, we get a shift reduce conflict

- $E \implies T \cdot +E$
- $E \implies T \cdot$

Without looking at next input, we are unsure whether to shift or reduce. Solutions is just to look ahead.

We can extend the item notation to include lookaheads which allows us to reduce conflicts. This becomes $LR(1)$

- $E \implies T \cdot +E : +$
- $E \implies T \cdot :\dashv$

$LR(k)$ DFA would use k lookaheads. The $LR(1)$ DFA can already get really big.

Solution: Add follow(E) to the reducible item and reduce using $E \implies T$ if the next input symbol is in follow(E).

This is called SLR(1), simplified LR(1) that produces a smaller DFA than LR(1) DFA

2.4 Semantics

Potential problems after parse tree

- Variable/procedure use with no declaration
- Duplicate variable/procedure
- Type checking
- Scopes

We will use context sensitive languages

- **Specify:** Context sensitive grammar
- **Recognition:** Linear bounded automata

Use tree traversal instead