CS 241
Course Review

Overview

1. We started by learning about binary representations of data: unsigned numbers, signed numbers, and characters.
2. Next we learned how to represent *program code* in binary using MIPS machine language. This was annoying, so our next goal was to write an *assembler* that converts human-readable MIPS assembly language into MIPS machine language.
3. We learned about regular languages, finite automata, and their use in *scanning*: splitting a program into small chunks of text called *tokens*. We used scanning to make it easier for our MIPS assembler to process the input program.

Overview

4. We learned how to write an assembler, and used our assembler to implement common features of high-level languages in MIPS assembly, such as arrays, conditionals, loops, and procedures.
5. We learned about *loading*, the process by which programs are placed in RAM, and *linking*, the process by which program modules can be combined with each other or with external libraries.

At this point, we had reached a solid understanding of how programs in a high-level language could be executed on a computer once expressed in assembly, but there was still a big gap in our knowledge: how to *translate* high-level programs into assembly programmatically.

Overview

6. We learned that regular languages are not suitable for describing the *syntax rules* of most high-level languages, since they cannot handle nested structures. We introduced *context free languages* to solve this problem. The corresponding *context free grammars* can be used to describe nested structures with a *parse tree*.
7. Finding a parse tree programmatically is difficult. We explored *top-down LL(1) parsing* but found its limitations too harsh.
8. We settled on using *bottom-up LR(1) parsing*. The parsing phase of compilation has the dual purpose of checking for syntax errors *and* building a parse tree for the program.

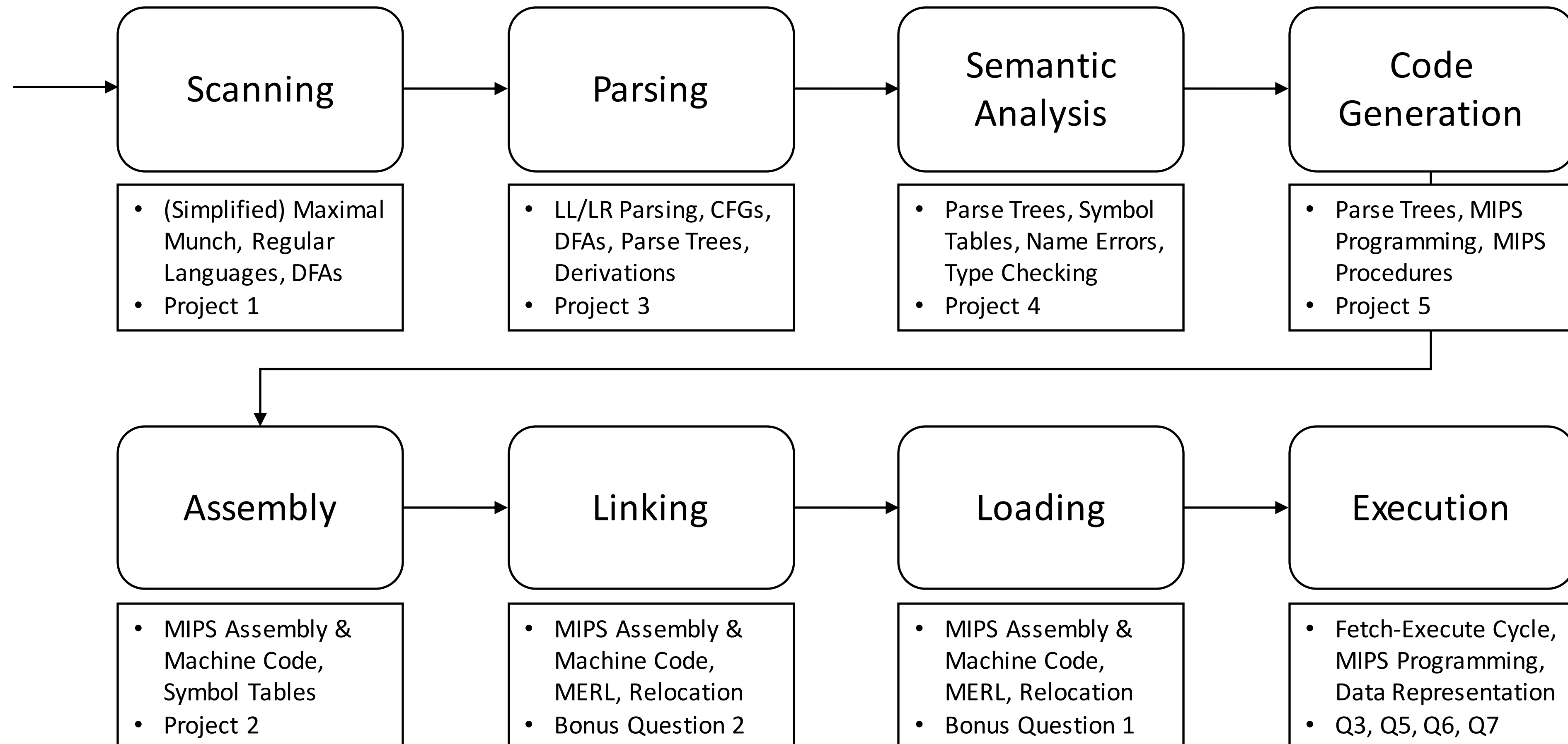
Overview

9. Parsing catches syntax errors, but cannot handle certain types of errors where the syntax is correct, but the *meaning* is invalid. We learned how to traverse the parse tree and perform *semantic analysis* to detect errors related to variable names and types, as well as collecting information about expression types.
10. After passing semantic analysis, the program is free of compile-time errors, and we can perform *code generation*. We learned how to traverse the parse tree and translate high-level C-like programming constructs into low-level MIPS assembly language.

Overview

11. To generate code for "new" and "delete", we relied on a pre-written memory allocation library. We explored how this library could be implemented, and also learned about automatic or "implicit" techniques for memory management (garbage collection).
12. In an optional section (for enrichment purposes) we discussed compiler optimization techniques that can be used to produce more efficient generated code.

Compiling and Executing a Program



Data Representations

Key concepts:

- Computers store everything using binary representations of data.
- The meaning of binary data depends on the chosen interpretation.
- Unsigned binary representation for non-negative integers.
- Two's complement binary representation for signed integers.
- Hexadecimal as a shorthand for binary data.
- The ASCII encoding system for English text.
- Bitwise operations for manipulating binary data.

Machine & Assembly Language

Key concepts (machine language):

- MIPS hardware: registers (general-purpose, PC, IR, hi, lo) and RAM (byte-addressed, word-aligned).
- Code is just binary data stored in RAM and has no distinction from other data except that it follows a specific format.
- The Fetch-Execute cycle for executing code stored in RAM.
- MIPS machine language and encodings of instructions.
- The distinction between MIPS assembly language and MIPS machine language.

Machine & Assembly Language

Key concepts (programming in assembly):

- Arithmetic in MIPS: add, sub, mult/multu, div/divu, mflo/mfhi.
- Loading data into registers with lis.
- Conditional branching and loops: Comparisons with slt/sltu, branching with beq/bne, using branches to create loops.
- Use of labels and how labels get translated into addresses/offsets.
- Accessing and modifying memory with lw/sw.
- Terminating a program with jr $31.

Machine & Assembly Language

Key concepts (MIPS procedures):

- Using jalr to call a procedure and jr $31 to return.
- How jalr works and why we need it (as opposed to using beq/bne/jr).
- Using memory as a stack to save and restore registers.
- Why saving and restoring registers is important.
- How procedures calling other procedures (including recursion) works by using the stack to save the registers for each call.
- Passing arguments and returning values using registers.

Regular Languages & Scanning

Key concepts:

- Formal language definitions: alphabet, string/word, language
- The empty string "" or ε (has length 0 and contains no symbols).
- Recursive definition of regular languages (union, concatenation, star) and regular expressions.
- Deterministic finite automata (DFAs): State diagrams, the DFA recognition algorithm.
- Nondeterministic finite automata (NFAs) and ε-transitions: How they work and how they simplify the description of some languages.
- Scanning: Maximal Munch and Simplified Maximal Munch algorithms.

Writing an Assembler

Key concepts:

- Why scanning to break the program into tokens is helpful.
- Error checking: syntax errors, range errors, duplicate label definitions, undefined labels, invalid instructions.
- Using a symbol table to keep track of defined labels.
- The need for two passes.
- Encoding instructions using bitwise operations.
- Producing output: why it's incorrect to print an ASCII string of "0" and "1" characters.

Loading & Linking

Key concepts:

- Why loading everything at address 0 is impractical.
- Why loading raw MIPS machine code at a non-zero address can cause problems, and why we need an object code format (MERL).
- Why we need a linker (as opposed combining code pre-assembly).
- The concepts underlying the MERL file format: why the three types of entries (REL, ESR, ESD) are needed for loading and linking.
- The correspondence between MIPS assembly source code and MERL: why and when each kind of entry gets generated.
- The loading/relocation and linking algorithms.

Context-Free Languages

Key concepts:

- The limitations of regular languages (can't handle arbitrarily deep nested structures, e.g., balanced parentheses).
- Context-free grammar definitions: terminals, nonterminals, productions, derivations, the language of a grammar.
- The relationship between derivations and parse trees.
- Ambiguity: A grammar is ambiguous if more than one [leftmost derivation, rightmost derivation, parse tree] exists for a word.
- Designing grammars to enforce operator precedence/associativity.

Top-Down Parsing

Key concepts:

- Augmenting grammars with beginning-of-file and end-of-file symbols.
- The high level idea behind top-down parsing.
- Using a Predict table to make decisions during parsing.
- Nullable, First and Follow: The intuitive ideas, and the algorithms.
- Using Nullable, First and Follow to fill out a Predict table.
- The definition of an LL(1) grammar.
- LL(1) limitations and ways to work around them, e.g., left factoring.

Bottom-Up Parsing

Key concepts:

- The high level idea behind bottom-up parsing.
- The notion of "items" and the LR(0) DFA construction.
- Using the LR(0) DFA to decide whether to shift or reduce.
- Shift-reduce and reduce-reduce conflicts.
- The SLR(1) DFA: How Follow sets resolve conflicts.
- The formal LR(1) parsing algorithm using stacks and a DFA.
- What it means for a grammar to be LR(0), SLR(1), etc.

Semantic Analysis

Also known as Context-Sensitive Analysis. Key concepts:

- Certain kinds of errors are not purely syntax-based, and require context from other parts of the program, so cannot be detected during parsing.
- Name errors (duplicate name errors, undefined name errors, scoping errors) can be detected using a symbol table.
- For WLP4, we need a "top-level" symbol table containing all procedures and "local" symbol tables for each procedure.
- Type errors require a system of type computation rules and type correctness rules.
- Type checking can be performed by applying these rules in a recursive tree traversal to annotate the tree with types.

Code Generation

Key concepts:

- The use of a frame pointer to find variables on the stack.
- Using the stack for temporary values when evaluating expressions.
- Calling procedures from external libraries.
- Ensuring label names are unique when generating branching code and procedure name labels.
- Using type information to generate different code for pointer arithmetic.
- Passing arguments and setting up the stack for procedures.
- Each *procedure call* needs its own frame pointer and its own set of local variables (to support recursion).

Memory Management

Key concepts:

- The free list algorithm for variable size blocks: allocation (splitting or removing a block from the free list), deallocation (inserting in the free list in sorted order, merging adjacent free blocks)
- Fragmentation and how it can arise. First/best/worst fit heuristics.
- The binary buddy algorithm: allocation (splitting the heap recursively to get an appropriately sized block), deallocation (merging the newly freed block with its buddy recursively).
- The high level ideas behind garbage collection algorithms: reference counting, mark and sweep, copying collection, generational GC.

Thank you, and good luck with the final exam!

The following slides have suggestions for how to practice each topic.

Photo by Tony Hisgett - Flickr: Geoffroy's Marmoset 1, CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=14621454

Data Representations: Practice

- If you are uncomfortable converting between binary, decimal, and hexadecimal, or between unsigned binary and two's complement binary, choose random small numbers and practice this.
- Identify which MIPS instructions have separate signed and unsigned versions. For each pair of instructions, find values where the signed and unsigned version of the instruction give different answers.
- Give examples of MIPS programs where a value is interpreted as something other than a signed or unsigned integer, such as:
  - A memory address.
  - An ASCII character.
  - A MIPS instruction.

Machine & Assembly Language: Practice

- Using the MIPS reference sheet, figure out which MIPS instructions are equivalent to small numeric constants. Find two MIPS instructions that can be added together to produce a different MIPS instruction.
- Write a MIPS program that actually performs this addition of instructions, and confirm that the result is the instruction you expect.
- Explain why the program on the left crashes, but the one on the right does not. What is the result in $3 for the right program?

```
div $0, $0
jr $31
```

```
lis $3
div $0, $0
jr $31
```

Machine & Assembly Language: Practice

- Implement the print procedure from Q6 in a different way. If you solved the problem using iteration, try to write a recursive solution; if you solved it with recursion, try an iterative solution.
- Implement an array sorting algorithm in MIPS, and use your array printing program to confirm that it works. Quadratic time, iterative algorithms like bubble sort will be the easiest to implement, but you can try a faster recursive algorithm like quicksort or merge sort if you want a challenge.
- Implement a recursive procedure that computes Fibonacci numbers (there's one on my slides, but try to do it without a reference).

Writing an Assembler: Practice

- The real version of MIPS has an "add immediate" instruction that lets you add a 16-bit constant value to a register.
  - Based on what you know about other MIPS instructions, what do you think the machine language encoding of this instruction would look like?
  - Make up an encoding for this instruction (or look up the real one if you want) and describe how to construct the encoding using bitwise operations.
- Write a short MIPS program that uses labels in various ways, and create the symbol table for this program. Use cs241.binasm and cs241.binview (or xxd) to confirm that your answer is correct.
  - If you don't know how to confirm your answer with these tools, figuring that out is itself a good exercise.

Scanning: Practice

- Come up with examples (scanning DFA and string to tokenize) for each of the following situations:
  - The string cannot be tokenized at all.
  - The string can be tokenized by Simplified Maximal Munch.
  - The string can be tokenized by Maximal Munch but not the simplified version.
  - It is possible to split the string into tokens, where each token is recognized by the scanning DFA. However, Maximal Munch fails to find a tokenization.
- There is an example in Chapter 3 of a case where Maximal Munch runs in quadratic time due to repeated backtracking. Reviewing this example might help with understanding the algorithm.

Regular Languages: Practice

- When you see a regular expression in the notes or slides, try to create an equivalent DFA or NFA, and vice versa. (This may not always be easy.)
- Look at Chapters 6, 7, and 8 of the notes for context-free grammars.
  - For each grammar, try to guess whether the language generated by the grammar is a regular language.
  - If you don't think it is regular, give your intuition as to why. (*Proving* a language is non-regular is outside the scope of the course.)
  - If you think it is regular, try to come up with a DFA, NFA, or regular expression for the language. If it seems impossible, it might be non-regular.
- Checking your answers to language problems can be tricky.
  - Try making two lists of short strings: Ones that are in the language, and ones that are not. Check each string against your DFA/NFA/RE.

Linking & Loading: Practice

- Write short MIPS programs and assemble them into MERL with cs241.linkasm. You can use "xxd -c4 *file*" *or* "cs241.binview -h *file*" to view the contents of a MERL or MIPS file in hexadecimal.
  - Create a program which has multiple REL entries. Use cs241.binview to confirm the presence of the REL entries. Then relocate the code segment of the MERL file to a different starting address. You can use the command "cs241.merl *address* < input.merl > output.mips" to check your answer.
  - Create two programs, Program A and Program B, for the purpose of linking. Program A should have two ESRs, one of which is resolved by linking and one which isn't, and an ESD. Program B should have an ESR that gets resolved, an ESD, and at least one REL entry. Link them by hand, then check your answer with cs241.linker.

Context-Free Languages: Practice

- Every regular language can be represented by a context-free grammar. Try creating grammars for the regular expressions, DFAs, and NFAs you see in the notes and slides.
- To practice derivations, look at grammars from the notes or slides. Try to find a string where a derivation requires you to use every rule in grammar. If this is not possible, try to find a minimal set of strings that uses all the rules.
- Write down a leftmost derivation, and draw the corresponding parse tree for the derivation. Determine whether the rightmost derivation corresponding to the parse tree is different from the leftmost one.

Context-Free Languages: Practice

- Take an unambiguous grammar from the notes or slides. Pick a string generated by the grammar and find the unique leftmost derivation. Is the unique rightmost derivation different? Try to find an example where it is. The parse tree will be the same for both derivations.
- Create an unambiguous grammar for arithmetic expressions with subtraction and division. Both operations should be left-associative. Division should have precedence over subtraction.
  - Switch the precedence of division and subtraction.
  - Switch the associativity of one or both operations from left to right.
  - Using short expressions, investigate how these changes affect parse trees.

Top-Down Parsing: Practice

- Take a context-free grammar that is *not* LL(1) (for example, any ambiguous grammar will work) and trace through the top-down parsing algorithm with a short string. You will need to decide which rules to apply through intuition. Pay attention to moments where you seem to have multiple valid rule choices.
- The following fun website can be used to compute Nullable, First, and Follow for a grammar, and if the grammar is LL(1), it can generate a Predict table. You can use this to double check your own computations, or to generate Predict tables for parsing practice.

https://smlweb.cpsc.ucalgary.ca/start.html

Bottom-Up Parsing: Practice

- The fun website from earlier can also generate LR(0) and SLR(1) automata for grammars, and even point out conflicts.

  https://smlweb.cpsc.ucalgary.ca/start.html

- Note that the conventions used by this website differ slightly from those in the course notes. If the LR(0) or SLR(1) DFAs you draw are not exactly the same as the ones this website generates, think carefully about whether you actually made a mistake, or if it is just a different convention for how things like the start and end of input are handled. Note also that the numbers assigned to states are totally arbitrary and don't mean anything; it is the items that matter.

Bottom-Up Parsing: Practice

- Ensure you can identify conflicts in LR(0) and SLR(1) DFAs, and distinguish between shift-reduce and reduce-reduce conflicts. Try to find or come up with examples of both kinds of conflicts.
- Ensure you can identify when Follow sets in an SLR(1) DFA resolve conflicts, and when they do not. Try to find or come up with examples of both cases.
- Practice tracing the LR(1) parsing algorithm using an SLR(1) DFA. Try tracing the parse in two ways: using both a state stack and symbol stack, or only using a symbol stack (in which case you need to run this stack through the DFA to determine whether to shift or reduce).

Semantic Analysis: Practice

- Come up with WLP4 programs that pass parsing, but are rejected during the semantic analysis phase, for each of the following reasons:
  - Duplicate identifier names.
  - Undeclared identifier, where the identifier is not present at all in the program.
  - Undeclared identifier, where the identifier is present but in a different scope.
  - Type error in an [addition, subtraction, multiplication] expression.
  - Error with [number of arguments, type of arguments] in a procedure call.
  - A type can be computed for each expression (meaning there are no type errors in expressions) but:
    - There is a type error in a statement.
    - There is a type error in something other than a statement.

Semantic Analysis: Practice

- Think about how you would handle semantic analysis for C features that are not in WLP4. Some examples to consider:
  - Assignment as an operator in expressions (rather than as a statement)
  - Bitwise operators
  - Comparison operators in expressions (rather than in tests only)
  - Declarations in the middle of a procedure
  - Explicit type conversion (casting)
  - For loops, do-while loops, switch statements
  - Global variables
  - Having different return types for procedures (not only int)

Code Generation: Practice

- Think about how you would implement code generation for C features that are not in WLP4. Some examples to consider:
  - Global variables
  - Declarations initialized to expressions (rather than constants)
  - Mid-procedure return statements
  - For loops, do-while loops, switch statements, break and continue in loops
  - Logical operators (&&, ||, !) with short-circuiting for && and ||
  - Increment and decrement (++, --), both pre- and post- versions
  - The ternary conditional operator (condition ? true-value : false-value)
- Consider features outside of C as well (though some may be difficult).

Memory Management: Practice

- Come up with an arbitrary pattern of allocations and deallocations and perform them using either the free list or binary buddy algorithm. Draw the heap after each allocation or deallocation.
- Consider the first fit, best fit, and worst fit heuristics for the free list algorithm. For each heuristic X, try to come up with a sequence of allocations and deallocations where heuristic X fails due to not having a large enough free block, but one of the other heuristics would work.
- For the binary buddy system, come up with a sequence of allocations and deallocations such that the final deallocation causes exactly 3 merges to happen, but there is still at least one allocated block.

Extending WLP4: Practice

- A great way to practice many concepts from the second half of the course is to extend WLP4 with a new feature.
- If you completed all the projects, you can actually implement these extensions.
  - Modify your scanning DFA in Project 1B to accept new tokens if necessary.
  - Modify the context-free grammar. There is a tool called **cs241.slr** that can generate SLR(1) DFA data for a CFG, which you can use with your Project 3 solution.
  - Modify your Project 4 solution to implement checks related to names or types.
  - Modify your Project 5 solution to generate code for the new feature.
- If you didn't complete the projects, you can still think about what changes you would need to make to the WLP4 specification.