

# Compilers Review

Rob Dickerson

Nov 5

- What is lexical, syntactic, and semantic analysis?
  - lexical analysis - recognize tokens of a language by reading a stream of characters from left to right and grouping them into tokens
  - syntactic analysis - recognize structure of language constructs (usually a tree) showing relationships
  - semantic analysis determines if constructs make sense in the language

# Why are context-free languages important for languages?

- CFGs can be translated into PDAs, and then to tables that can be used by the compiler
- Easy to extend a language that has a CFG

# What can't CFGs capture?

- $L = \{w cw \mid w \text{ is in } (0+1)^*\}$  which corresponds to identifiers being declared before they are used.
- $L = \{a^n b^m c^n d^m \mid n, m \geq 1\}$  which can be interpreted as procedure declarations  $a$  and  $b$  with parameters  $n$  and  $m$ , and procedure calls of  $c$  and  $d$  with the same number of parameters.

# What is a symbol table?

- A symbol table holds information related to the various identifiers in a program.
- Stores attributes:
  - storage allocated
  - its type
  - scope
  - procedures - type and calling method

# Why do we use a symbol table?

- Symbol information is not context-free and cannot be determined using simple parsers
- There is a lack of locality in that the location where a symbol is defined is rarely where it is used.

# What is a problem that separate compilation causes for type checking?

- In separate compilation, type information needs to be kept until link time to verify that parameter types to functions match as well as types of external variables.

# Describe some limitations that a one-pass compiler forces upon a language

- Variables and functions must be defined before their use
- Tends to lead to programs read bottom-up. Such as “main” declared last.

# What is “backpatching”?

- In some languages variables are allowed to be used before they are declared and most have support for “goto”.
- Such constructs are difficult to implement using a one-pass compiler.
- It is possible to leave a blank slot that will be filled in later when the information becomes available.

# Outline of a compiler

- Lexical analysis
- Syntactic analysis
- Semantic analysis
- Intermediate code generation
- Optimization
- Code generation
- Linking and loading

# Lexical analysis

- Source is broken into lexical items.
  - identifiers, operation symbols, punctuation, keywords, comments, etc.
- The identifiers are loaded into the symbol table.
- Lexical items are often converted into an internal form.

# Syntactic analysis

- A formal grammar is used to parse the program and build a tree

# Semantic analysis

- Symbol table is updated - adding type information
- Implicit information is inserted such as resolution of overloading operators, error detection
- tree is traversed to generate intermediate code

# Intermediate code generation

- Some generate this explicit format for an abstract machine
- Must be both easy to generate and easily modified into the final code
- (End of the front-end)

# Optimization

- The code is modified to run faster and or save space without modifying semantics

# Linking and loading

- Object code is collected (from all the separately compiled units and libraries).
- All external references in modules are resolved and translated into locations relative to the beginning of the program.
- Program is then loaded into some memory location and its addressing done relative to that base address

# What is the difference between bottom-up and top-down parsing?

- Bottom-up: one starts with the input and tries to reduce it the start symbol of the language
- Top-down: One begins with the top-symbol of the language and tries to find a derivation of the input

# Why avoid left-recursion for top-down parser?

- It is possible to get into an infinite loop, in which the parser continually calls the recursive procedure because the input does not change.
- ex:  $\text{expr} \rightarrow \text{expr} + \text{term}$
- All recursion can be eliminated by producing an equivalent right-recursive grammar.

# How does a LL parser recognize languages?

- a LL parser is predictive.
- Assume the input is an instance of the language and then attempts to predict based on the next few characters (left to right).
- Predictive parsing is a type of recursive descent where a lookahead symbol unambiguously determines the procedure selected for non-terminal.

# How about the LR parser?

- Reads and stacks characters until it recognizes the right hand side of the production on the top of the stack.
- Then, reduces the stack by replacing the characters that it recognized with the left hand side of the production.
- When the parser runs out of input, the only thing left on the stack should be the start symbol.

# What are the relative benefits of predictive and reductive parsers?

- Predictive parser: easy to write by hand, resulting tables are generally smaller, but the grammar must be able to disambiguate productions using lookahead symbol.
- Reductive: efficiently recognize a larger class of languages, and better at handling errors.
- Some languages lend themselves to one

# What is a shift-reduce conflict?

- Occurs in a LR parser when at a particular state it cannot decide whether to shift another input symbol onto the stack or to reduce the symbols currently on the stack.

# Why are AST's used?

- Allow for some independence between the front and back ends of the compiler.
- Provide an easy way to do manipulation or optimization of the program.

# What's the difference between a l-value and r-value?

- An lvalue is a part of an expression that represents a location.
- A rvalue is an expression that yields a value.
- Think of an assignment statement: left is address and right is a value

# 5 compiler optimizations that save time

- Common subexpression removal
- Loop invariant code motion
- Strength reduction
- Inlining
- Code motion
- Pipeline scheduling

# 5 optimizations for space

- Induction variable elimination
- Constant propagation
- Copy propagation
- Stack height reduction
- Register allocation

# What is dataflow analysis?

- Static analysis of the possible use of variables by a program when it executes.
- Information is useful when determining optimizations to be performed.

# What machines are required for lexical and syntactic analysis?

- Lexical analysis: finite state machine
- Syntactic analysis: PDA