CS202 Compiler Construction

February 27, 2003

Nature of midterm:

Open book/notes (anything you want to bring)

Questions will be conceptual; reasoning, not writing code

Will try to abstract from the details of our compiler (as much as possible)

Midterm topics:

Regexp
Finite automata
Lexers
Tokens
Jlex (automated lexer builders)
Grammars
Derivations
Parsers:
LL(1) (aka top-down aka predictive)
LR(k) (aka bottom-up aka shift-reduce)
## Midterm topics:

- Jcup (automated parser generators)
- Error recovery
- Parse Trees: concrete and abstract
- Symbol tables
- Pond parse tree generation
- Type systems
- Pond static type analysis
- Activation frames
- Call stacks

---

### Lexers are specified using regular expressions

Lexers are implemented as finite automata (FA)
- Edge for each letter in alphabet
- Some states are accepting

Fact: Regexps describe regular languages, finite automata (FA) recognize regular langs.

---

### NFA: nondeterministic finite automata

DFA: deterministic finite automata

Fact: Any regexp can be automatically converted to an NFA

Fact: Any NFA can be converted to a DFA

DFAs more efficient
Grammars describe context-free languages

PL syntax is context-free

Productions map non-terminals to sequence of terminals and non terminals

\[ e \rightarrow e + e \]
\[ e \rightarrow e - e \]
\[ e \rightarrow n \]

Every grammar has a start symbol

A sentence is a sequence of non-terminals

A sentential form is a sequence of terminals and non-terminals that can occur in a legal derivation

A derivation is a sequence of derivation steps

Each derivation step replaces a single non-terminal \( s \) with sentential form \( \gamma \), if \( s \rightarrow \gamma \) is in grammar

A sentence is in a language if there exists a derivation of it, given grammar

Example:

\[ e \rightarrow e + e \rightarrow e + e - e \rightarrow e + e - 2 \]
\[ e + 3 - 2 \rightarrow 1 + 3 - 2 \]

Note: rightmost derivation

Leftmost derivation of same sentence:

\[ e \rightarrow e - e \rightarrow e + e - e \rightarrow 1 + e - e \]
\[ 1 + 3 - e \rightarrow 1 + 3 - 2 \]

Note: multiple derivations do not imply ambiguous grammar
Recognizers decide whether a sentence is in a language defined by grammar

Parsers are recognizers that additionally generate parse trees

Recognizers/parsers reconstruct derivations of input sentences

LL parsing: reads tokens left-to-right, reconstructs leftmost derivation top-down

LR parsing: reads tokens left-to-right, reconstructs rightmost derivation bottom-up

Simplest parsing is LL(1)

Places limitations on grammar; no left recursion

Left recursion can be removed by left factoring

LL(1) grammar parsed by recursive descent or table-driven parser

also called top-down parsing

Recursive descent defined with big switch statement

One function per non-terminal

Every case is token

Need FIRST set for each token
LL(1) table driven parsers more efficient, defined via FIRST and FOLLOW sets

Let A be non-terminal in a grammar

FIRST(γ) is set of terminals that can begin sentential forms derivable from γ

FOLLOW(A) is set of terminals that can immediately follow A in sentential forms

LR(1) grammars less restrictive

Parsed by shift-reduce parsers

Shift pushes input tokens onto stack
Reduce matches TOS to RHS of prod
  Pops off match
  Pushes on lhs of production

2 kinds of conflicts for s/r parsers

Shift/reduce conflicts:
  Can shift on another token, or reduce top of stack
  Cannot tell which from next token
    Both may be valid

Reduce/reduce conflicts
  can reduce two productions
Parse trees: generated during parse, represent grammatical structure

*Concrete* parse trees are faithful representations of derivations
Parse trees always rooted with start symbol
Read from left to right, leaves are all tokens
Each parent/children subtree is one production

*Abstract* parse trees eliminates “junk”
Preferable for PL compilation

```plaintext
eexpr ::= add_expr;
add_expr ::= add_expr PLUS mult_expr;
add_expr ::= mult_expr;
mult_expr ::= mult_expr STAR constant;
mult_expr ::= constant;
constant ::= INT_CONST;
```

1 + 2 + 3 * 4 is implicitly (1 + 2) + (3 * 4):
If more than one concrete parse tree valid for any sentence, grammar is ambiguous

Lack of definition of precedence or associativity introduces most ambiguity

Precedence / associativity controlled by grammar or rules

Precedence rules list operators in precedence order
Each rule indicates associativity
for left, right, non-associative

Grammars chain for precedence
Use left recursion for left associative
Right recursion for right associative

Symbol tables
Store information about identifiers, e.g.:
Name
Type
Source code position
We use them for static type checking
Implemented as hash tables, with opns.:
Insert
Lookup
Extend (for entering new scope)
Retract (for leaving scope)
Parse tree generation:

Concrete parse trees generated by constructing program objects in actions
Concrete parse trees represent syntactic constructs
Trees constructed as part of error recovery

Type systems:

*Static* type systems enforce run-time safety

Type safety ideal: type-checked programs are free of run-time errors.

Note: run-time error not a logical (programmer error)

Example of possible run-time error:

```c
int i, y;
int *x = &5;
for (int i = 0; i < 10000; i++) x++;
y = x + 3; // what's the result?
```

Another example: using undeclared variable.
Type checking: checks that declared types of expressions is consistent

Type inference: reconstructs types for type-annotation-free programs

No “true” type system: different systems, different benefits/drawbacks

Pond type system:

Uses notion of subtype compatibility to achieve flexibility for programs

*Parse-time* type checking: type analysis performed during parsing

*Post-parse* type checking: type analysis separate pass on completed parse trees

Activation Frames:

At run time, store information relevant to function calls
  - Function parameters
  - Local variables
  - Return address

A recursive function can have many activations
Call stacks:

- Data structure for organizing activation records during runtime
- LIFO structure corresponds to call sequences
- Push on call, pop on return

Bookkeeping during function calls:

- Values may be stored in stack or in registers
- Registers much faster, but practically finite
- When new function called, old values must be saved

Callee-save registers
- Saved by called function
Caller-save registers
- Caller saves

No clear division of responsibility; choice defined by architecture and convention
General convention:

If values are to be maintained for use after function call, use callee-save
Callee saves all

If some values can be discarded upon function call, use caller-save
Caller can select which to save

Use together to minimize saving