# CSE401: Target Code Generation and Midterm

David Notkin
Autumn 2000



## Codegen example

```
void IfStmt::
        codegen(SymTabScope* s, RegisterBank* rb) {
  Reg test = _test->codegen(s, rb);

  char* elseLabel = TheAssembler->newLabel();
  TheAssembler->branchFalse(test, elseLabel);
  rb->freeReg(test);

  for (int i = 0; i < _then_stmts->length(); i++) {
    _then_stmts->fetch(i)->codegen(s, rb);
  }

  TheAssembler->insertLabel(elseLabel);
}
```



```
void CallStmt::
        codegen(SymTabScope* s, RegisterBank* rb) {
  for (int i = _args->length() - 1; i >= 0; i--) {
    Expr* arg = _args->fetch(i);
    Reg areg = arg->codegen(s, rb);
    TheAssembler->push(areg);rb->freeReg(areg);
  }
  SymTabScope* enclScope;
  SymTabEntry* ste = s->lookup(_ident, enclScope);
  …
  Reg staticLink = s->getFPOf(enclScope, rb);
  TheAssembler->push(staticLink);
  rb->freeReg(staticLink);
  rb->saveRegs(s);
  TheAssembler->call(_ident);
  rb->restoreRegs(s);
  TheAssembler->popMultiple((_args->length() + 1) *
                        TheAssembler->wordSize());
}
```



## Another example

```
void AssignStmt::
       codegen(SymTabScope* s, RegisterBank* rb) {
  int offset;
  Reg base = _lvalue->codegen_address(s, rb, offset);

  Reg result = _expr->codegen(s, rb);

  TheAssembler->store(result, base, offset);

  rb->freeReg(base);
  rb->freeReg(result);
}
```



## Next lecture

- Run-time system calls
- Beginning of optimization
  - There is none in PL/0 (either basic or extended)



## Part I

- If G is an ambiguous context-free grammar, then there is at least one sentence in the language defined by G that has two possible parse trees.
- It is possible to define a lexer in terms of a context-free grammar.
- Semantic type-checking is necessary for ensuring that a break statement in enclosed within a loop statement.
- A handle in bottom-up parsing corresponds to the intersection of the FIRST and FOLLOW sets for a non-terminal in a grammar.
- Strong typing would preclude a program from adding two integers and storing them in a pointer to an integer.

# Part IIa

- Describe a language that can be recognized by a context-free grammar but not a deterministic finite-state machine.
- Lisp is a _____ and _____ typed language.
- To programming recursively in a language that does not explicitly support recursion, you have to implement your own _____.
- In a language like PL/0, constant values are usually stored in the _____.



# Part IIb

- If you have two grammars for the same language, where one grammar is LL(1) and the other is LR(1), would you expect the semantic checking for these grammars to be substantially the same or significantly different? Briefly justify your answer.
- Give an example of a legality check on input programs that could be made using a context-free grammar but that is generally more practical to check during semantic analysis.
- Concisely distinguish between overloading and polymorphism.



# Part IIc

- Which is more expensive to apply, structural equivalence or name equivalence of types? In one sentence, justify your answer.
- Concisely address the following statement: "If you have an ambiguous context-free grammar, then a rightmost derivation will always differ from a leftmost derivation."
- We prefer for our internal representations to capture abstract syntax. Briefly: why, then, do we have concrete syntax?



# Part III

- The C programming language has a large number of operators (for defining expressions) with 16 levels of precedence. The first five levels of the precedence structure are:
  - Parentheses
  - Structure access `. ->`
  - Unary `! - ++ -- * &`
  - Multiply, divide, mod `* / %`
  - Add, subtract `+ -`
- Furthermore, all the binary operators associate left-to-right, while the unary operators (including structure access) associate right-to-left (this isn't exactly the way that C does associativity, but use it for this problem). Write a context-free grammar that defines expressions using these operators with this precedence and associativity structure.



# Part IV

- Compute the FIRST and FOLLOW sets for the following grammar:

  G ::= ABCabcd
  A ::= a | ε
  B ::= b | ε
  C ::= c | ε

- State the precise relationship between FIRST and FOLLOW sets and predictive parsing.



# Part Va

- Briefly explain the role, if any, that a symbol table must play in supporting dynamic scoping.
- Assume you are compiling a language that supports concurrency (i.e., multiple threads of control). Concisely discuss the following statement: "Because there are multiple threads of control, the stack-discipline used in managing the symbol table during compilation (for languages like PL/0) must be completely redesigned."

# Part Vb

Some languages, such as Pascal, have a `with` statement such as:

```
var new_patient: Patient;
    old_patient: Patient;        …
   with new_patient: new, old_patient: old do begin
       new.LastName := 'Smith';
       new.FirstName := 'Abby';
       new.Sex := Female;
       old.LastName := 'Brown';
       old.FirstName := 'Henry';
       old.Sex := Male
   end;
```

Basically, with allows you to use a different (usually shorter) name for record instances. For example, inside the with clause you can write new.FirstName to refer to the FirstName field of record new_patient, because of the binding in the with clause. What changes, if any, are needed to a PL/0-ish compiler to accommodate a with statement.