CSC 425 - Principles of Compiler Design I

A Simple Compiler

The ac Compiler

- The source language is ac (adding calculator) from the Crafting a Compiler textbook
- The target language is dc (desktop calculator), a reverse-polish desk calculator.

The ac Grammar

1	Prog	::= Dcls Stmts $
2	Dcls	::= Dcl Dcls
3		| λ
4	Dcl	::= floatdcl id
5		| intdcl id
6	Stmts	::= Stmt Stmts
7		| λ
8	Stmt	::= id assign Val Expr
9		| print id
10	Expr	::= plus Val Expr
11		| minus Val Expr
12	Val	::= id
13		| inum
14		| ifnum

ac Tokens

Terminal	Regular Expression
floatdcl	`f`
intdcl	`i`
print	`p`
id	`[a-e] | [g-h] | [j-o] | [q-z]`
assign	`=`
plus	`+`
minus	`-`
inum	`[0-9]+`
fnum	`[0-9]+.[0-9]+`
blank	`(" " | "\n" | "\t")+`

Compiler Phases for ac

- Lexer (Scanner)
  - Partition the source text into tokens
  - Associate semantic values with id, inum, and fnum tokens
  - Check for illegal tokens
- Parser
  - Verify that the program conforms to the ac syntax
  - Build an abstract syntax tree (AST)
- Semantic Analyzer
  - Build a symbol table
  - Check for duplicate declarations
  - Type check assignment statements
  - Annotate the AST with additional information
- Code Generation
  - Generate dc code by traversing the annotated AST

Example ac Program

- Source text:

```
f b
i a
a = 5
b = a + 3.2
p b
```

- Tokens (semantic values are in parentheses):
  floatdecl, id(b), intdecl, id(a), id(a), assign, inum(5), id(b), assign, id(a), plus, fnum(3.2), print, id(b), $

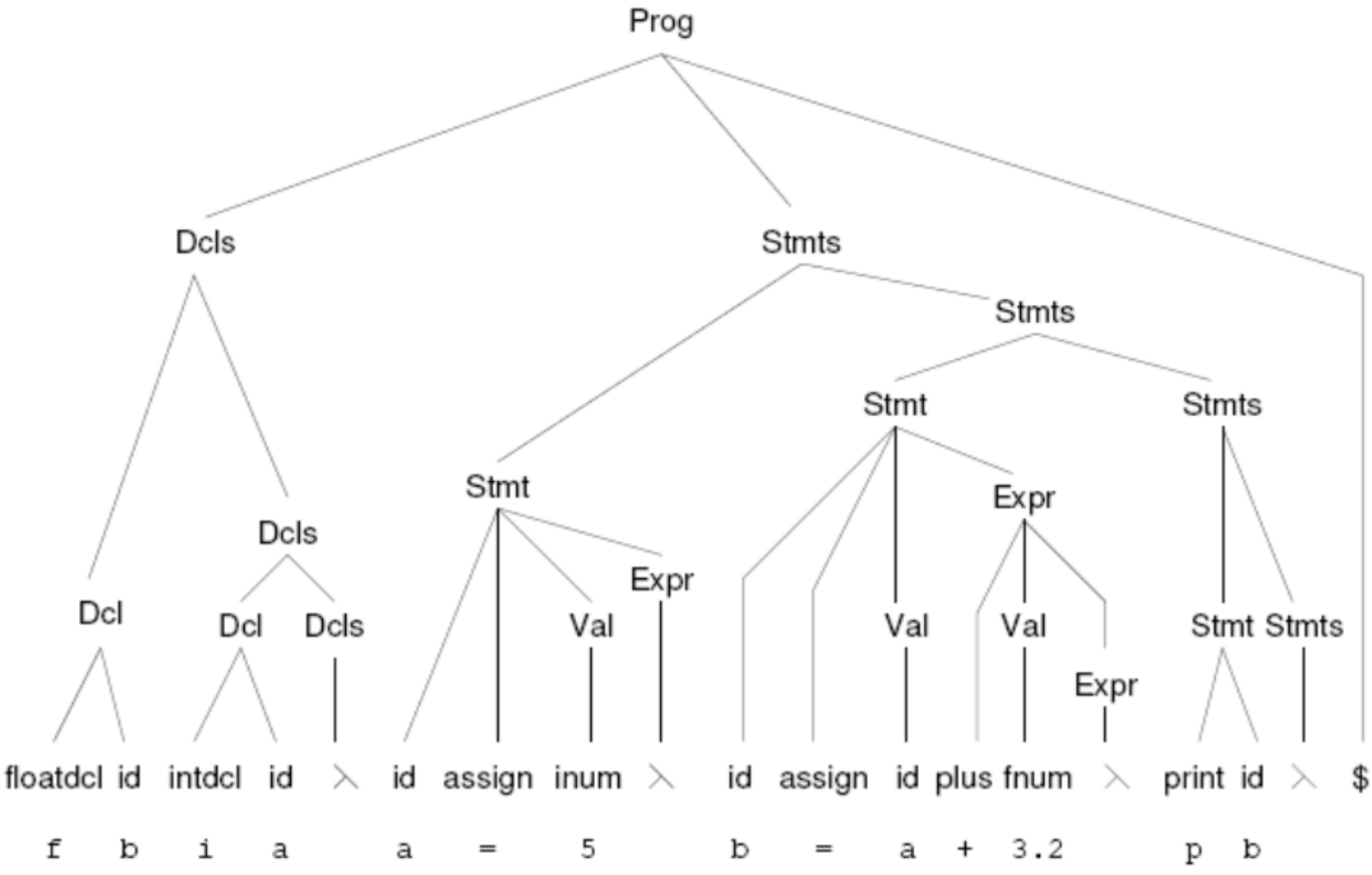
Example Program Derivation

Rule	Sentential Form
	⟨ Prog ⟩
1	**⟨ Decls ⟩ Stmts $**
2	**⟨ Dcl ⟩ Dcls** Stmts $
4	**floatdcl id** ⟨ Dcls ⟩ Stmts $
2	floatdcl id **⟨ Dcls ⟩ Dcls** Stmts $
5	floatdcl id **intdcl id** ⟨ Decls ⟩ Stmts $
3	floatdcl id intdcl id ⟨ Stmts ⟩ $
6	floatdcl id intdcl id **⟨ Stmt ⟩ Stmts** $
8	floatdcl id intdcl id **id assign ⟨ Val ⟩ Expr** Stmts $
14	floatdcl id intdcl id id assign **inum** ⟨ Expr ⟩ Stmts $

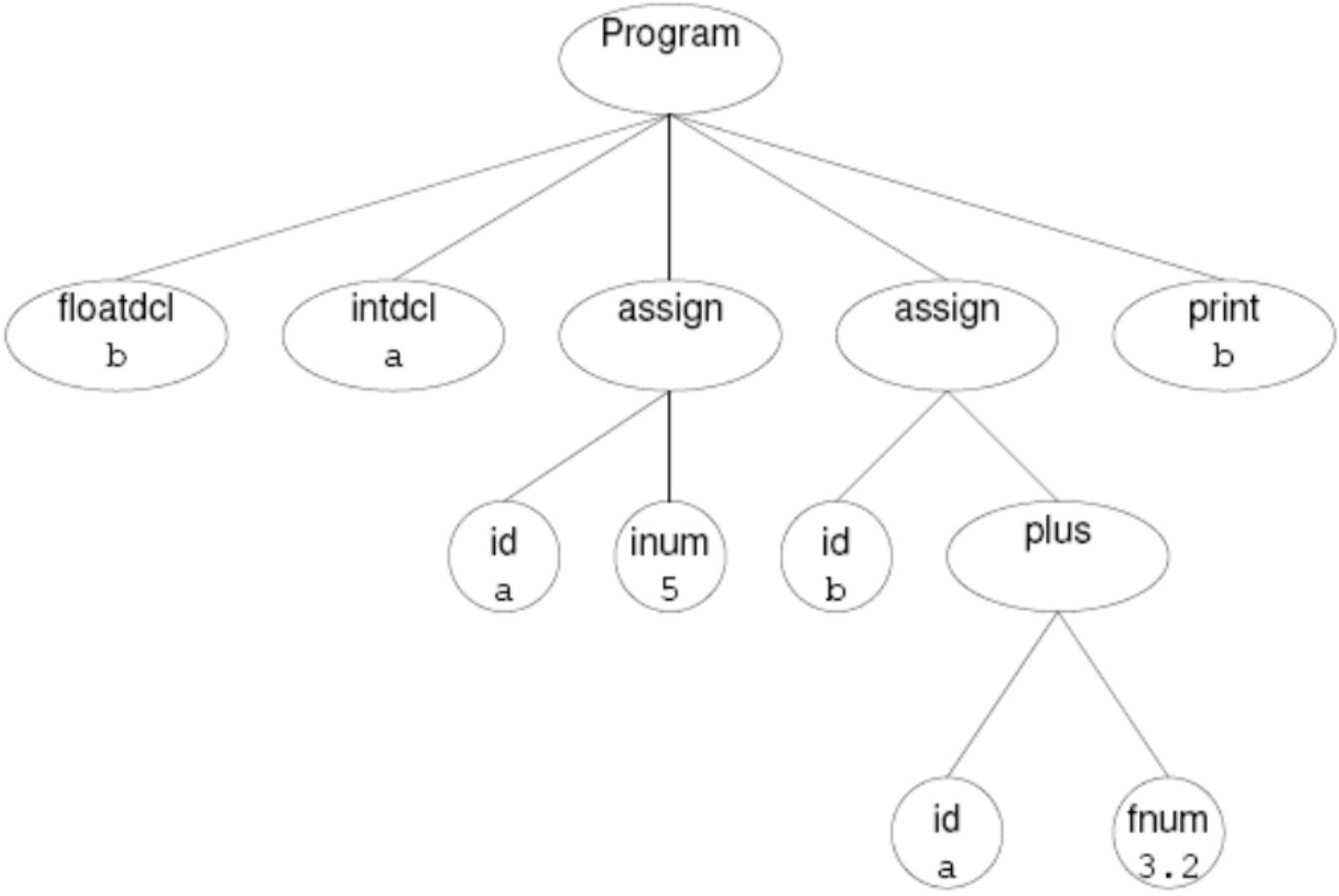
Example Program Derivation Continued

Rule	Sentential Form
12	floatdcl id intdcl id id assign inum ⟨ Stmts ⟩ $
6	floatdcl id intdcl id id assign inum **⟨ Stmt ⟩ Stmts** $
8	floatdcl id intdcl id id assign inum **id assign ⟨ Val ⟩ Expr** Stmts $
13	floatdcl id intdcl id id assign inum id assign **id** ⟨ Expr ⟩ Stmts $
10	floatdcl id intdcl id id assign inum id assign id **plus ⟨ Val ⟩ Expr** Stmts $
15	floatdcl id intdcl id id assign inum id assign id plus **fnum** ⟨ Expr ⟩ Stmts $
12	floatdcl id intdcl id id assign inum id assign id plus fnum ⟨ Stmts ⟩ $
6	floatdcl id intdcl id id assign inum id assign id plus fnum **⟨ Stmt ⟩ Stmts** $
9	floatdcl id intdcl id id assign inum id assign id plus fnum **print id** ⟨ Stmts ⟩ $
7	floatdcl id intdcl id id assign inum id assign id plus fnum print id $

Example Program Parse Tree



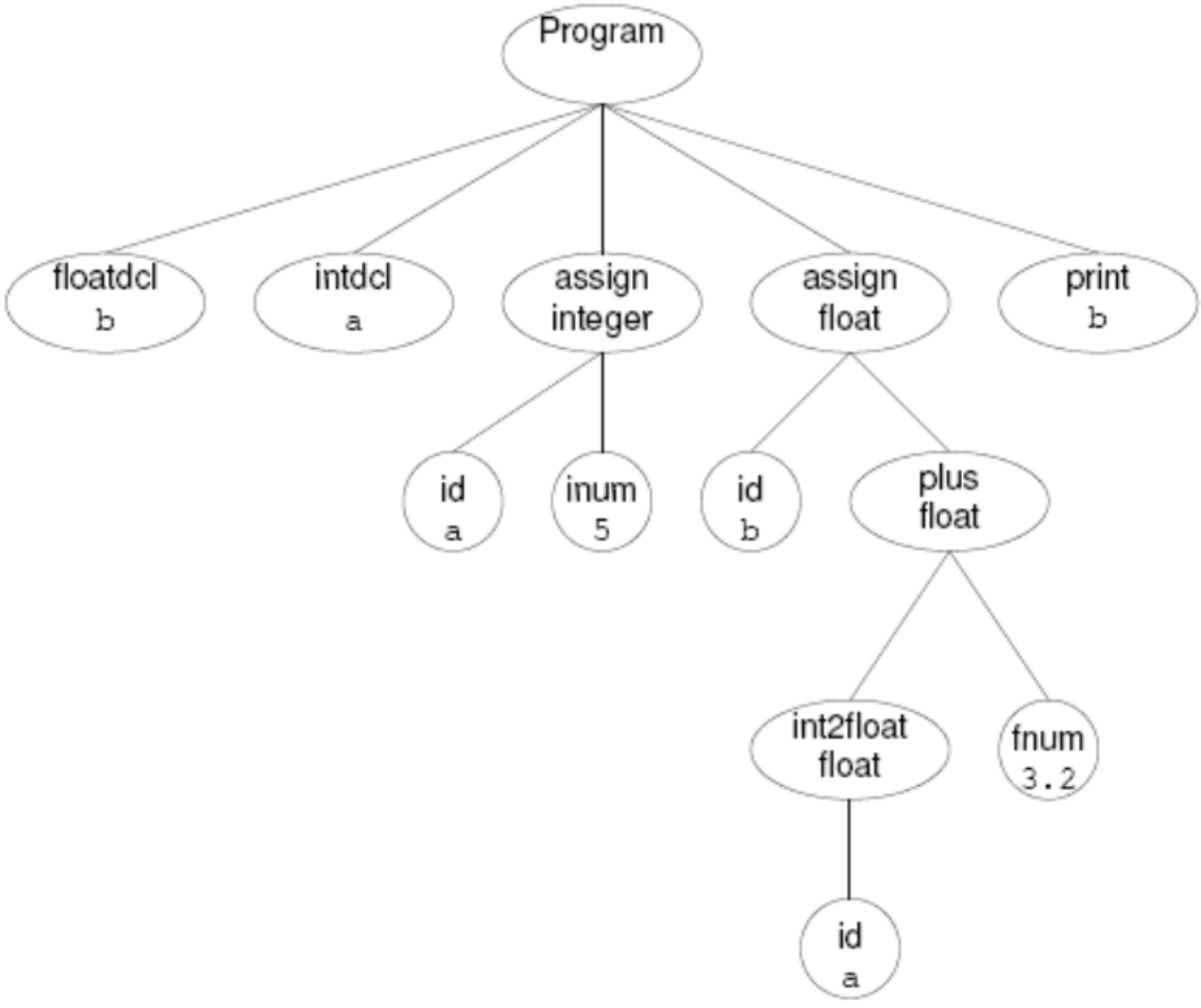
Example Program Abstract Syntax Tree



Example Program Symbol Table

Symbol	Type	Symbol	Type	Symbol	Type
a	integer	k	null	t	null
b	float	l	null	u	null
c	null	m	null	v	null
d	null	n	null	w	null
e	null	o	null	x	null
g	null	q	null	y	null
h	null	r	null	z	null
j	null	s	null		

Example Program Annotated AST



Example Program Code Generation

Source	Code	Comments
`a = 5`	`5`	Push 5 on stack
	`sa`	Pop stack; store value in register `a`
	`0 k`	Reset precision to integer
`b = a + 3.2`	`la`	Load register `a`; push value to stack
	`5 k`	Set precision to float
	`3.2`	Push 3.2 on stack
	`+`	Add (pop operands; push result)
	`sb`	Pop stack; store value in register `b`
	`0 k`	Reset precision to integer
`p b`	`lb`	Load register `b`; push value to stack
	`p`	Print top of stack
	`si`	Pop stack using store to `i` register