CSC 448: Compilers

Lecture 1

Joseph Phillips
De Paul University

2015 April 3

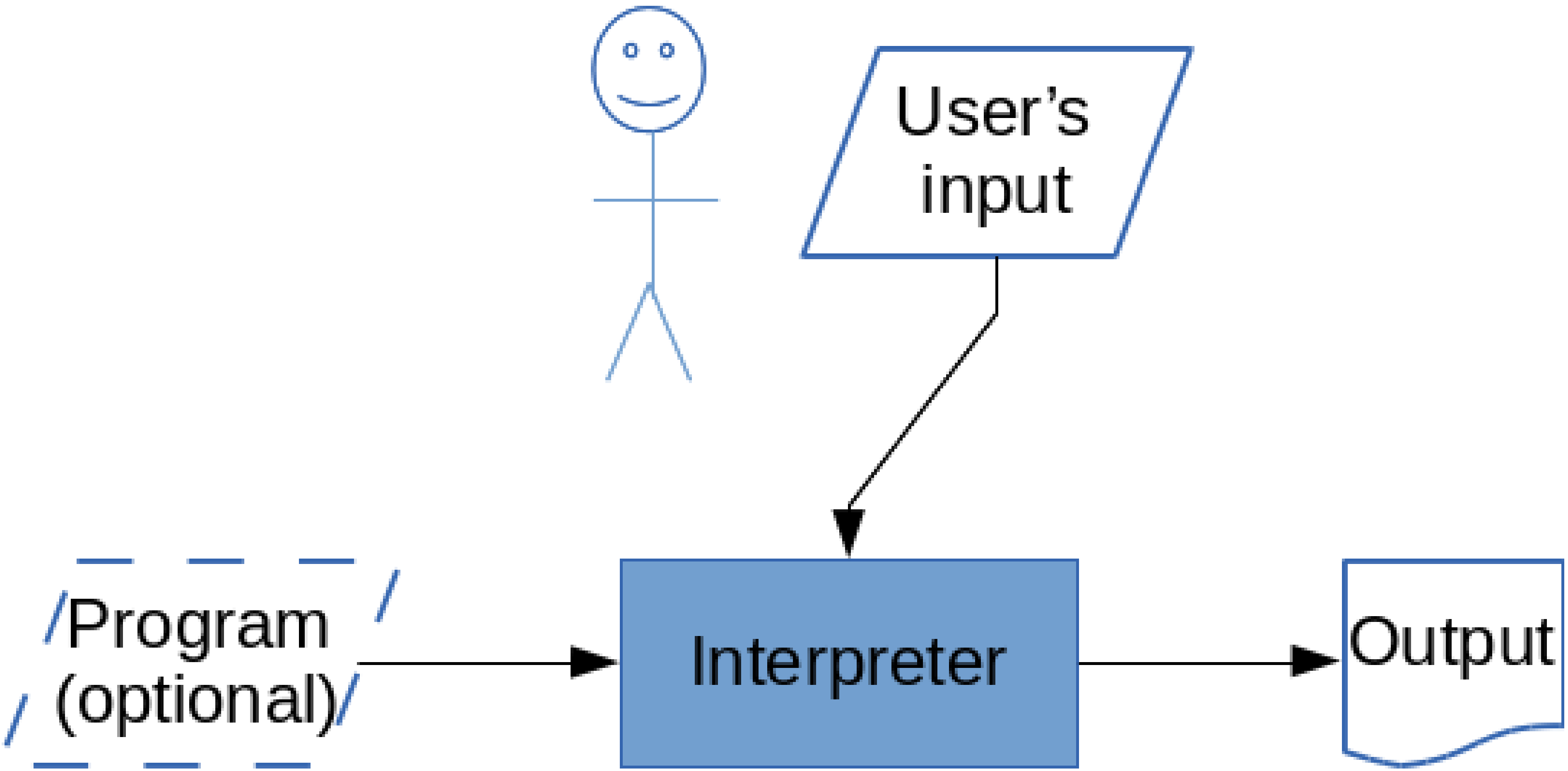
Reading

- Charles Fischer, Ron Cytron, Richard LeBlanc Jr. “Crafting a Compiler” Addison-Wesley. 2010.
  - Chapter 1: Introduction
  - Chapter 2: A Simple Compiler

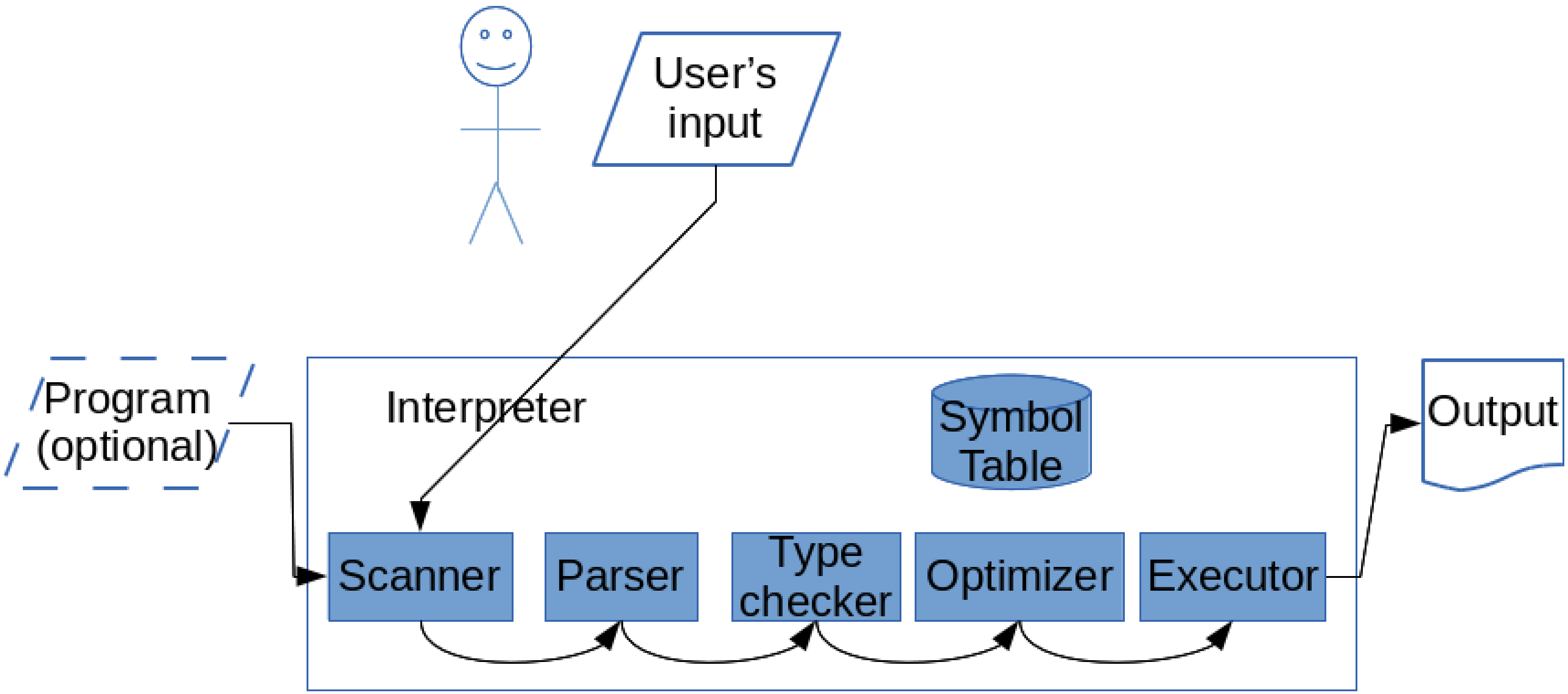
Topics:

- Introduction
  - Motivation:

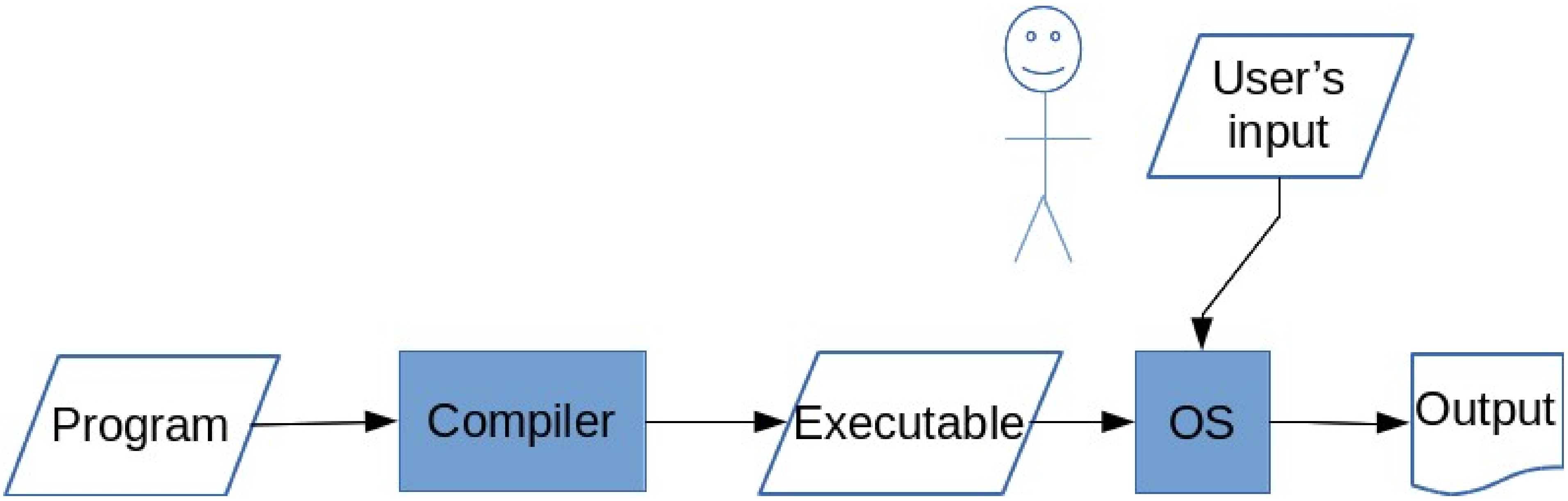
Programmer's View of Interpreted Language



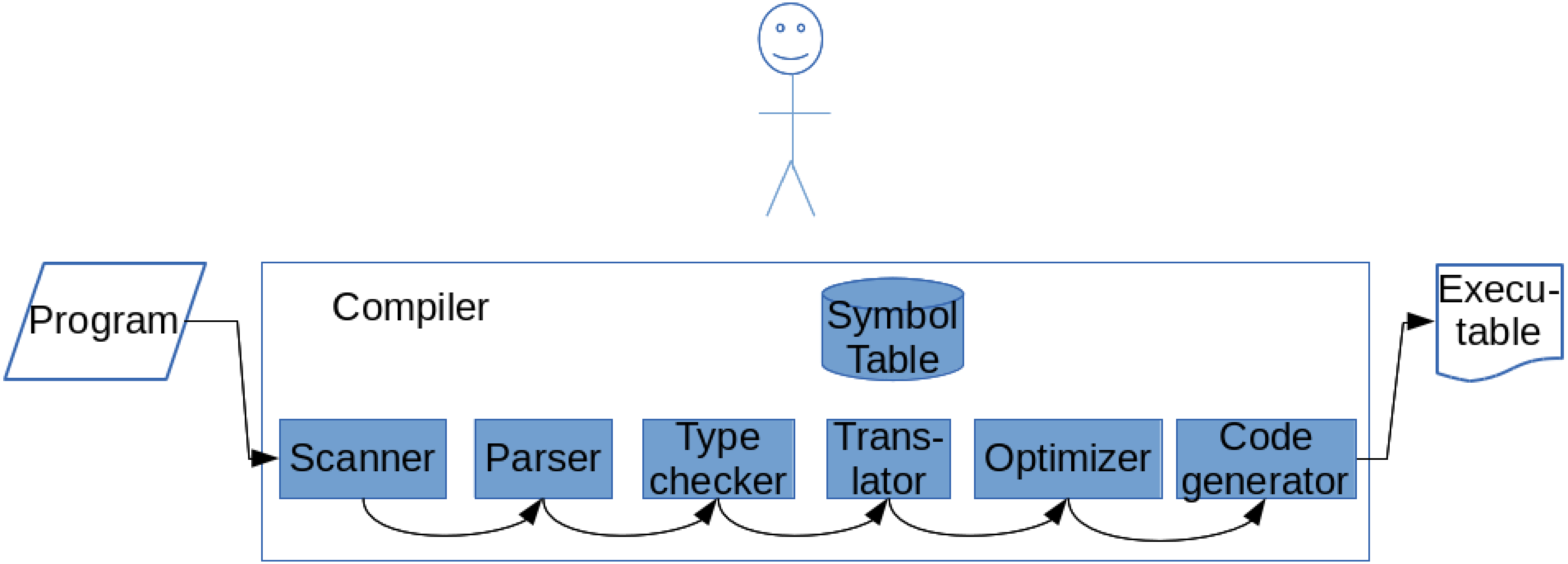
. . . and underneath the hood



Programmer’s view of compiled language



. . . and underneath the hood



Take home lesson . . .

- Even if you ***never*** write a “compiler”, ***this compiler class will help you*** write programs that interpret user input
  - Command ***line prompts***
  - Reading ***file formats*** -> turn into ***data-structures***
  - ***Write your own language!***

So, what are these parts?

- **Scanner**: two parts
  - Characters -> “lexemes” (strings)
  - “lexemes” -> tokens (data structures rep. parts of program)
- **Parser**
  - Builds data-structure (generally tree) corresponding to structure of program
- **Type checker**
  - Check types (silly)
- **Translator**
  - Translates data structure to new format (*e.g*. assembly language)

So, what are these parts?

- **Optimizer**
  - Gets rid of inefficiencies in new code
- **Code generator**
  - Write new code
- **Symbol table**
  - Keeps track of user-defined “symbols” (functions, variables, types, classes, *etc*.)
  - Stores info like:
    - Name
    - Return type,
    - Parameter types
    - Location in source code (for debugging)

But first . . . recursion on trees!

1) Write a C structure with:

- A “left” pointer
- A “right” pointer
- Data (like a string)

2) Write a recursive-function to the nodes:

- Prefix order
- Postfix order
- Infix order

And second . . . defining grammars with productions

- A way to:
  - ***Specify a grammar*** for a language
  - ***Generate grammatical*** (but perhaps nonsensical) sentences in that language can be generated
- Let's use ***simplified English*** as an example:
- **Terminals**: symbols that mean specific strings in the language.
  - `Det` (determiner): "the", "a", "these", *etc*.
  - `N` (noun): "man", "hat",
  - `V` (verb): "saw", "ate"
  - `Adj` (adjective): "hungry", "blind", "felt"

And second . . . defining grammars with productions (2)

- **Non-Terminals**: symbols that group zero or more terminals underneath them into grammatical structures.
  - `S` (sentence)
  - `NP` (noun-phrase)
  - `VP` (verb-phrase)
  - `AdjL` (adjective list)

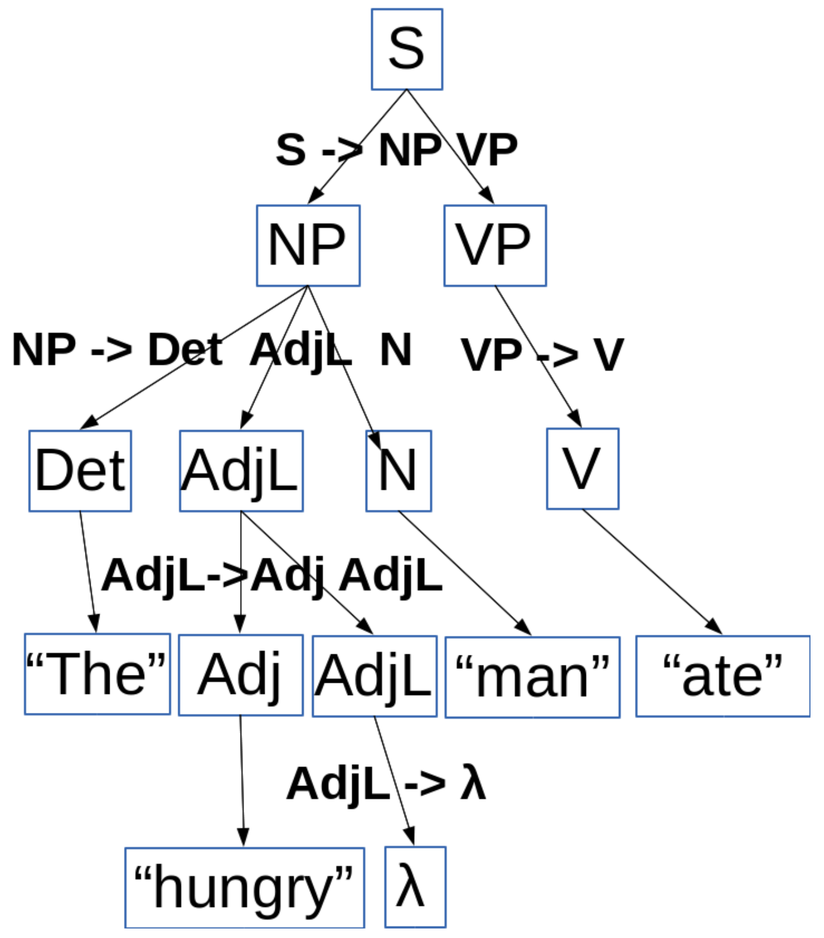
And second . . . defining grammars with productions (3)

- **Productions**: re-write rules that tell how non-terminals can be expanded into a ***list of zero or more*** terminals and non-terminals.
- To be useful, each non-terminal should have at least one production

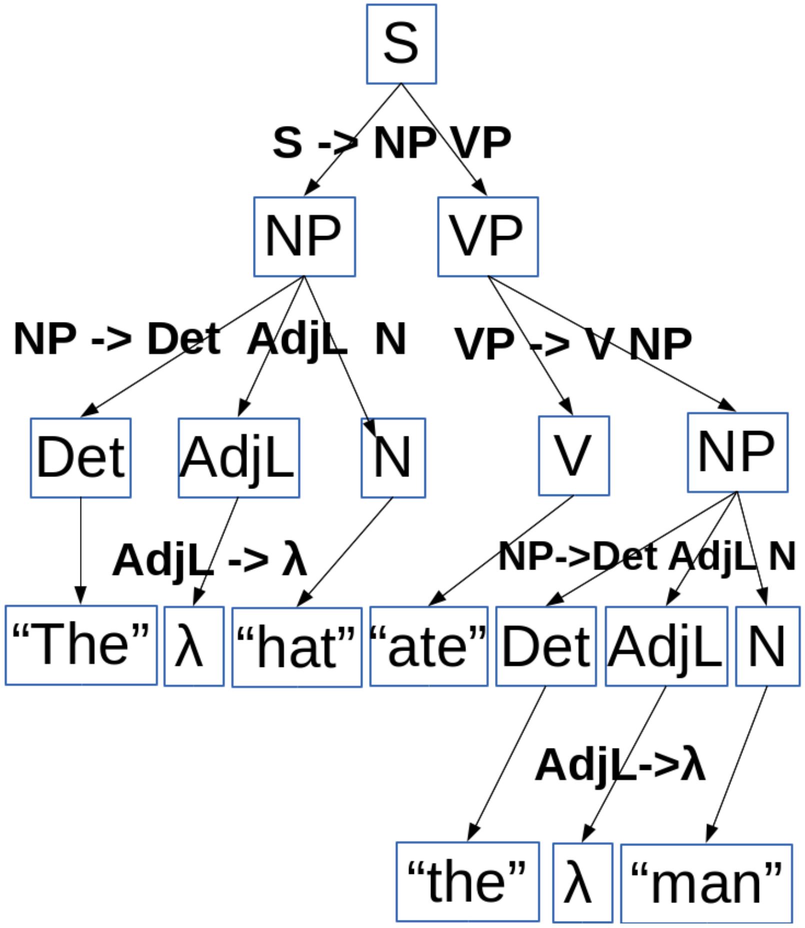
```
S     -> NP VP
NP    -> Det AdjL N
VP    -> V
       |  V NP
AdjL  -> Adj AdjL
       |  λ
```

- λ means ”empty string”

And second . . . defining grammars with productions (4)



And second . . . defining grammars with productions (5)



And now, our parser

- **Input** (the "**ac**" language):
  - A **simple arithmetic language** with:
    - Variables
    - Addition and subtraction
- **Output** (the "**dc**" language):
  - A simple RPN (= ***Reverse Polish(*) Notation*** = postfix) Unix calculator
  - "2 * (3 + 4)" represented as "2 3 4 + *"
  - ***Mathematicians***: ***No need for parentheses!***
  - ***Computer Scientists***: Easy for ***stack-based machines*** to calculate
- * "**RPN**" is ***not*** meant as an ethnic slur!
  - English-speakers do not know how to pronounce the name of 20th Century Polish mathematician ***Jan Łukasiewicz*** who invented ***prefix notation*** ("* + 3 4 2")

And now, our parser: the grammar

```
Prog   -> Dcls Stmts $
Dcls   -> Dcl Dcls
       |  λ
Dcl    -> 'f' id
       |  'i' id
Stmts  -> Stmt Stmts
       |  λ
Stmt   -> id '=' Val Expr
       |  print id
Expr   -> '+' Val Expr
       |  '-' Val Expr
       |  λ
Val    -> id
       |  inum
       |  fnum
```

- $ means "end of input"
- λ means "empty string"

- Toy input language (left):
- A program:

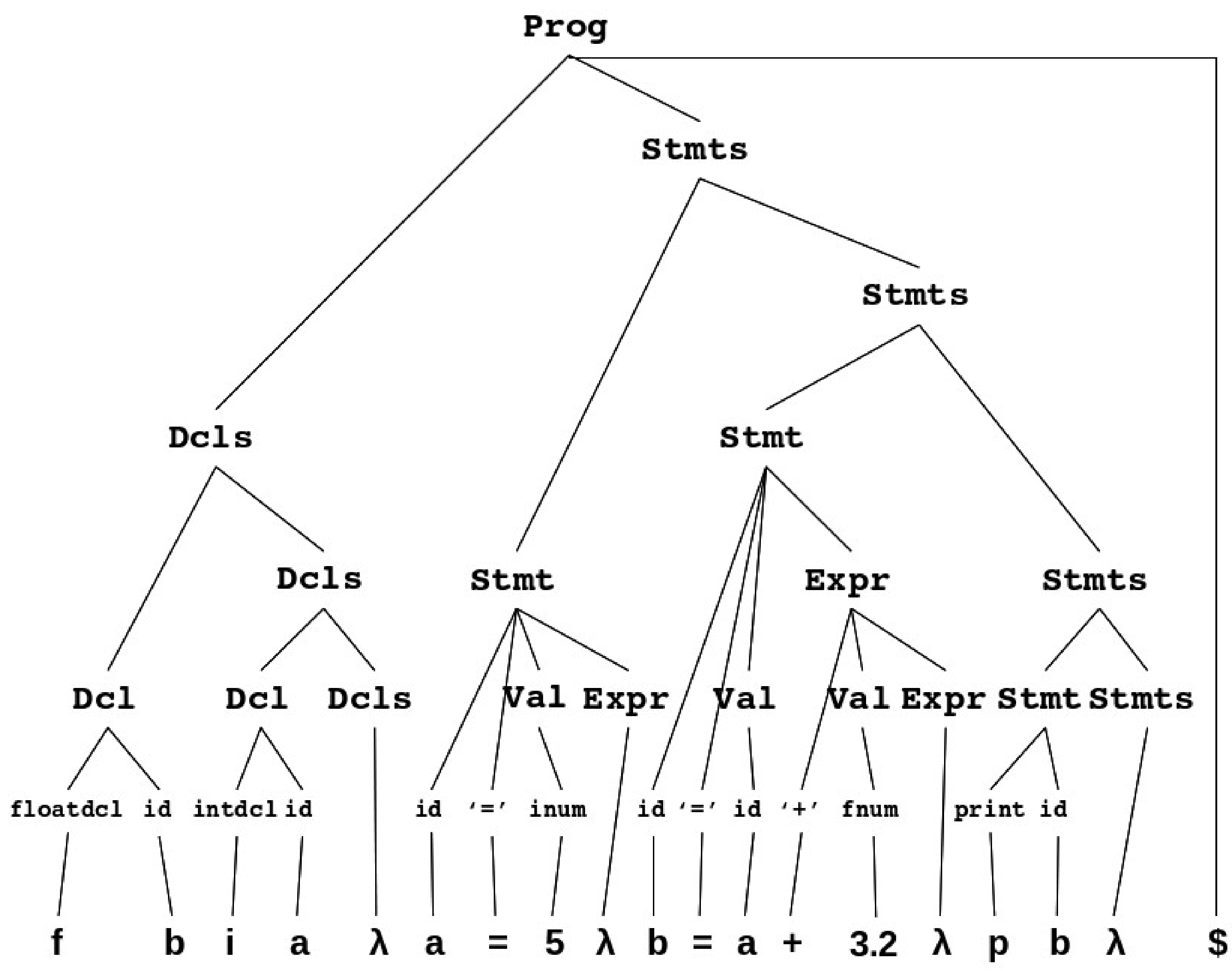
```
f b   # float var b
i a   # int var a
a = 5
b = a + 3.2
p b   # print b
```

- Or as, they would say:

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

How the grammar generates:

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



And now, our parser: the output language (1)

- Toy output language: dc (a stack-based calculator)

  Pushing operator:

  - Numbers (`5`)
  - (L)oad single-char register (`la`)

  Popping operators:

  - (S)tore single-char register (`sa`)

  Computing operators (pop, pop, compute, push result)

  - Addition (`+`)
  - Subtraction (`-`)

  Stack examining ops

  - (P)rint the top of the stack (without popping) (`p`)

And now, our parser: the output language (2)

```
$ dc
5
6
+
p
11
q
```

```
$ dc
5
sa
6
sb
la
lb
+
p
11
q
```

Scanning: character stream to lexemes

- Class `TokenStream`
  - Public interface:
    - constructor
    - `peek()`
    - `advance()`
  - Private worker methods
    - `scanner()`
    - `scanDigits()`

Parsing (1)

- Recursive descent
  - Start at first non-terminal of language: `Prog`
  - Structure of parsing routines follows structure of productions
  - Grammar:

```
Prog -> Dcls Stmts $
```

  - Code:

```
parseProg()
{
  parseDeclares();
  parseStatements();
  expect(END_OF_FILE_SYMBOL);
}
```

Parsing (2)

- Code expects only certain tokens (anything else is an error):

```
if  ( (tokenStream.peek() == FLOAT_DECLARE_SYMBOL) ||
      (tokenStream.peek() == INT_DECLARE_SYMBOL)   ||
      (tokenStream.peek() == ID_SYMBOL)            ||
      (tokenStream.peek() == PRINT_SYMBOL)         ||
      (tokenStream.peek() == END_OF_FILE_SYMBOL)
    )
{
  parseDeclares(tokenStream);
  parseStatements(tokenStream);
  expect(tokenStream,END_OF_FILE_SYMBOL);
}
else
  throw "expected floatdcl, intdcl, id, print, or eof";
```

Parsing (3)

- Code **generates a data-structure** that **corresponds to user's program**'s structure
- **Returns** this data-structure:

```
Symbol* symbolPtr;
. . .
parseDeclares(tokenStream);
symbolPtr=parseStatements(tokenStream);
expect(tokenStream,END_OF_FILE_SYMBOL);
. . .
return(symbolPtr);
```

Parsing (4)

- Putting it all together:

```
Symbol* parseProg (TokenStream& tokenStream)
{
  Symbol* symbolPtr;

  if  ( (tokenStream.peek() == FLOAT_DECLARE_SYMBOL) ||
        (tokenStream.peek() == INT_DECLARE_SYMBOL)   ||
        (tokenStream.peek() == ID_SYMBOL)             ||
        (tokenStream.peek() == PRINT_SYMBOL)          ||
        (tokenStream.peek() == END_OF_FILE_SYMBOL)
      )
  {
                 parseDeclares(tokenStream);
    symbolPtr  = parseStatements(tokenStream);
                 expect(tokenStream,END_OF_FILE_SYMBOL);
  }
  else
    throw "expected floatdcl, intdcl, id, print, or eof";

  return(symbolPtr);
}
```

Your turn:

Q1: The function `parseDeclares()` ***does not return anything***, why not?

Q2: Does `parseDeclares()` ***change any parser data-structure***?

Parsing (5)

- ***But wait!*** There's more!

```
parseDeclare()
parseDeclares()
parseValue()
parseExpression()
parseStatement()
parseStatements()
```

- Helper function(s):

```
expect()
```

Consistency (e.g. type) Checking

- `checkConsistency()`
  - Makes sure types agree for:
    - Assignments
    - Addition/subtraction
  - Helpers:
    - `getType()`: What is the type of this node?
    - `convert()`: Inserts in-between “conversion-node” if needed

Optimizing

Take Prof Joe’s Computer Systems II Class!

Code Generation

- `outputForDC()`
- What type of node is it?
  - Push-operation
  - Pop-operation
  - Calculating-operation
    - Two pops, calc, push
  - Examining operation
  - Precision-specification operation