Frontend: Lexing & Parsing

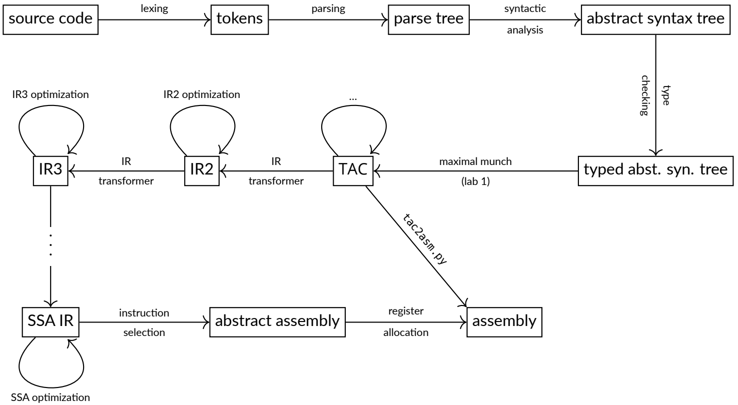
(CSC 3F002 EP) Compilers

Pierre-Yves Strub

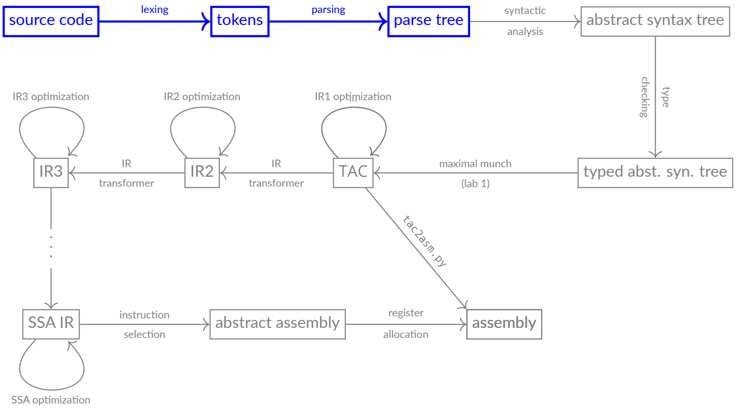
(Slide deck author: **Kaustuv Chaudhuri**)

2025-09-04 // Week 1

Compiler Stages: Block Diagram



Today's Focus
(will be part of lab 1)



BX Grammar (Straightline Fragment)

(we'll use this for the first labs)

```
⟨program⟩ ::= "def main() {" ( ⟨vardecl⟩ | ⟨stmt⟩)* "}"

⟨vardecl⟩ ::= "var" ⟨ident⟩ "=" ⟨expr⟩ ": int;"

⟨stmt⟩ ::= ⟨assign⟩ | ⟨print⟩
⟨assign⟩ ::= IDENT "=" ⟨expr⟩ ";"
⟨print⟩ ::= "print(" ⟨expr⟩ ");"

⟨expr⟩ ::= IDENT | NUMBER
         | ⟨expr⟩ "+" ⟨expr⟩ | ⟨expr⟩ "-" ⟨expr⟩
         | ⟨expr⟩ "*" ⟨expr⟩ | ⟨expr⟩ "/" ⟨expr⟩ | ⟨expr⟩ "%" ⟨expr⟩
         | ⟨expr⟩ "&" ⟨expr⟩ | ⟨expr⟩ "|" ⟨expr⟩ | ⟨expr⟩ "^" ⟨expr⟩
         | ⟨expr⟩ "<<" ⟨expr⟩ | ⟨expr⟩ ">>" ⟨expr⟩
         | "-" ⟨expr⟩ | "~" ⟨expr⟩
         | "(" ⟨expr⟩ ")"

IDENT ::≈ /[A-Za-z][A-Za-a0-9_]*/        (except reserved words)
NUMBER ::≈ /0|[1-9][0-9]*/               (value must fit in 63 bits)
```

Compiler Frontend: Reading the Source

- Steps:
  1. Convert BX0 source code (text) to token sequence (lexing)
  2. Parse token sequence (parsing)
  3. generate an abstract syntax tree (AST)
  4. Perform syntactic correctness checks on the AST
  5. Perform type checking to compute and store type information
  6. (Optional) Simplify the AST when necessary
     (e.g., convert `for` loops to `while` loops)
- Lexing and Parsing
  - Can be written by hand in e.g. Python
    - Flexible – can parse very complicated grammars
    - Labor intensive, esp. for maintenance
    - Difficult to make them fast
  - Lexer/parser generators transform the grammar into efficient lexers/parsers
    - Python: the PLY library
    - C/C++: (F)Lex and Yacc (or Bison)
    - Java: Antlr

Step 1: Lexical Scanning

- Source code contains many kinds of things:
  variables, numbers, operators, keywords, whitespace,
  comments, punctuation, file inclusion, …
- When defining a grammar, it makes sense to categorize these kinds of things into tokens
  - By convention, tokens are written in `UPPERCASE`
  - All variables are the same token: `IDENT` (“identifier”)
  - Each keyword is a separate token: `PRINT`, `IF`, `WHILE`, …
  - Operators and punctuations are also separate tokens: `PLUS`, `STAR`, `SEMICOLON`, `LPAREN`, `RPAREN`, …
  - Usually not tokens: whitespace, comments
- A lexical scanner (lexer) converts source code into tokens

A Simple Scanner

```
import re

ident_re = re.compile(r’[A-Za-z_][A-Za-z0-9_]*’)
wsp_re   = re.compile(r’[ \t\f\v\r\n]+’)

def lex(source, pos=0):
    while pos < len(source):
        # using Python 3.8+ syntax:
        if (match := wsp_re.match(source, pos)):
            pos += len(match.group(0))        # skip whitespace
        elif (match := ident_re.match(source, pos)):
            ident = match.group(0)
            yield (’IDENT’, ident, pos)
            pos += len(ident)
        else:
            print(f’Unknown character at offset {pos}: {source[pos]}’)
            pos += 1                          # skip it
```

```
>>> print(*lex(’mary had a little lambda’), sep=’\n’)
(’IDENT’, ’mary’, 0)
(’IDENT’, ’had’, 5)
(’IDENT’, ’a’, 9)
(’IDENT’, ’little’, 11)
(’IDENT’, ’lambda’, 18)
>>>
```

Adding Numbers

```python
# ...
number_re = re.compile(r'[0-9]+')

def lex(source, pos=0):
    while pos < len(source):
        # ...
        elif (match := number_re.match(source, pos)):
            numlit = match.group(0)
            yield ('NUMBER', int(numlit), pos)
            pos += len(numlit)
        # ...
```

```
>>> print(*lex('mary had 3 lambdas'), sep='\n')
('IDENT', 'mary', 0)
('IDENT', 'had', 5)
('NUMBER', 3, 9)
('IDENT', 'lambdas', 11)
>>>
```

Handling Line-Comments

```
# ...
comment_re = re.compile(r'//.*\n?')

def lex(source, pos=0):
    while pos < len(source):
        # ...
        elif (match := comment_re.match(source, pos)):
            pos += len(match.group(0))
        # ...
```

```
>>> print(*lex('mary had 3 // snow white\nlambdas'), sep='\n')
('IDENT', 'mary', 0)
('IDENT', 'had', 5)
('NUMBER', 3, 9)
('IDENT', 'lambdas', 25)
>>>
```

Tracking Line Numbers

```
# ...
wsp_re = re.compile(r’[ \t\f\v\r]+’) # r’\n’ is gone!
nl_re = re.compile(r’\n’)

def lex(source, line=1, pos=0):
    while pos < len(source):
        # ...
        elif nl_re.match(source, pos):
            pos += 1
            line += 1
        # ...
        elif (match := ident_re.match(source, pos)):
            ident = match.group(0)
            yield (’IDENT’, ident, line, pos)
            pos += len(ident)
        # ...
```

```
>>> print(*lex(’// a pythonic rhyme\nmary had 3 // snow white\nlambdas’), sep=’\n’)
(’IDENT’, ’mary’, 2, 20)
(’IDENT’, ’had’, 2, 25)
(’NUMBER’, 3, 2, 29)
(’IDENT’, ’lambdas’, 3, 45)
>>>
```

Operators and Punctuation

```python
# ...
lparen_re    = re.compile(r'\(')
rparen_re    = re.compile(r'\)')
plus_re      = re.compile(r'\+')
minus_re     = re.compile(r'-')
semicolon_re = re.compile(r';')

def lex(source, line=1, pos=0):
    while pos < len(source):
        # ...
        elif lparen_re.match(source, pos):
            pos += 1
            yield ('LPAREN', None, line, pos)
        elif rparen_re.match(source, pos):
            pos += 1
            yield ('RPAREN', None, line, pos)
        elif plus_re.match(source, pos):
            pos += 1
            yield ('PLUS', None, line, pos)
        elif minus_re.match(source, pos):
            pos += 1
            yield ('MINUS', None, line, pos)
        elif semicolon_re.match(source, pos):
            pos += 1
            yield ('SEMICOLON', None, line, pos)
        # ...
```

Handling Keywords

```
# ...
reserved = {
    'print': 'PRINT',
    'while': 'WHILE',
}

def lex(source, line=1, pos=0):
    while pos < len(source):
        # ...
        elif (match := ident_re.match(source, pos)):
            ident = match.group(0)
            yield (reserved.get(ident, 'IDENT'), ident, line, pos)
            pos += len(ident)
        # ...
```

```
>>> print(*lex('print(x + y);'), sep='\n')
('PRINT', 'print', 1, 0)
('LPAREN', None, 1, 6)
('IDENT', 'x', 1, 6)
('PLUS', None, 1, 9)
('IDENT', 'y', 1, 10)
('RPAREN', None, 1, 12)
('SEMICOLON', None, 1, 13)
>>>
```

Using PLY/Lex To Streamline Lexical Scanners

In a file called, say, `scanner.py`:

```
import ply.lex as lex

reserved = {’print’: ’PRINT’, ’while’: ’WHILE’}

# The ‘tokens’ tuple must be present and list all the valid tokens
tokens = (
  ’PLUS’, ’MINUS’, ’SEMICOLON’, ’LPAREN’, ’RPAREN’, ’IDENT’, ’NUMBER’
) + tuple(reserved.values())

# Regexp strings definitions beginning with ‘t_’ define simple tokens
t_LPAREN = r’\(’
t_RPAREN = r’\)’
t_PLUS = r’\+’
t_MINUS = ’-’
t_SEMICOLON = ’;’

# Functions beginning with ‘t_’ define complex token processing code.
# The docstrings of the functions contain the regexp that is matched for the token
def t_IDENT(t):
    r’[A-Za-z_][A-Za-z0-9_]*’   # docstring contains the regexp
    t.type = reserved.get(t.value, ’IDENT’)
    return t

def t_NUMBER(t):
    r’[1-9][0-9]*’
    t.value = int(t.value)
    return t
```

Using a PLY/Lex Scanner

Still in `scanner.py`:

```
# error handling with t_error()
def t_error(t):
    print(f”Illegal character ’{t.value[0]}’ on line {t.lexer.lineno}”)
    t.lexer.skip(1) # skip one character

# characters to ignore
t_ignore = ’ \t\f\v’

def t_newline(t):
    r’\n’
    t.lexer.lineno += 1
    # no return, signifying ignore

lexer = lex.lex()
# This will use Python introspection (reflection) to find out all the
# ‘tokens’ and ‘t_stuff’ in this module and create a suitable lexer from it
```

```
>>> from scanner import lexer
>>> lexer.input(’print(x + y);’)        # give it some input
>>> lexer.token()                       # get next token
LexToken(PRINT, ’print’, 1, 0)
>>> lexer.token()
LexToken(LPAREN, ’(’, 1, 5)
>>> t = lexer.token()
>>> (t.type, t.value, t.lineno, t.lexpos)
(’IDENT’, ’x’, 1, 6)
```

Step 2: Parsing

- Using the token stream generated by the lexer, the parser will convert it into an abstract syntax tree (AST)
- The structure of the AST is given by a grammar, usually specified in Backus-Naur Form (BNF).
- You have already seen BNF grammars in CSE 206.
- Example:

$$
\begin{aligned}
\langle \text{expr} \rangle ::= {} & \texttt{IDENT} \mid \texttt{NUMBER} \\
& \mid \langle \text{expr} \rangle \ \texttt{PLUS} \ \langle \text{expr} \rangle \mid \langle \text{expr} \rangle \ \texttt{MINUS} \ \langle \text{expr} \rangle \\
& \mid \texttt{LPAREN} \ \langle \text{expr} \rangle \ \texttt{RPAREN}
\end{aligned}
$$

Using PLY/Yacc to Build a LR Parser

In a file called, say, `parser.py`:

```
import ply.yacc as yacc
import bxast                    # We will write this module later

from scanner import tokens

## Every parser function is written with a ‘p_’ prefix.
## The docstring of the function is the portion of the grammar it handles

def p_expr_ident(p):
    """expr : IDENT"""
    #  p[0]    p[1]
    #
    # The parser function must define p[0] in terms of p[1], p[2], ...
    p[0] = bxast.Variable(p[1])

def p_expr_number(p):
    """expr : NUMBER"""
    p[0] = bxast.Number(int(p[1]))

def p_expr_plus(p):
    """expr : expr PLUS expr"""
    #  p[0]   p[1] p[2] p[3]
    p[0] = BinopApp('PLUS', p[1], p[3])

def p_expr_minus(p):
    """expr : expr MINUS expr"""
    p[0] = BinopApp('MINUS', p[1], p[3])

def p_expr_parens(p):
    """expr : LPAREN expr LPAREN"""
    p[0] = p[2]
```

Using the PLY/Yacc Parser

Still in `parser.py`:

```
parser = yacc.yacc(start='expr')
# Build the parser by introspection from all the ‘p_’ functions.
# The ‘start’ parameter determines the start symbol, i.e., the overall
# output of the parser
```

```
>>> from scanner import lexer
>>> lexer.input('(x + y)')        # give it some input
>>> from parser import parser
>>> p = parser.parse(lexer=lexer)
>>> p
<bxast.BinopApp object at 0x7f57550e5310>
>>> print(p.to_json())           # assuming you wrote such a function
[ [ "<binop>", [ [ "<var>", "x" ], [] ],
  [ [ "PLUS" ], [] ], [ [ "<var>", "y" ], [] ] ], [] ]
```

Operator Precedence

- Some operators can be used in multiple modes.
  E.g.: `-` can be used *infix* (subtraction) or *prefix* (negation)
- Infix operators have an associativity.
  E.g.: `(x - y + z)` is `((x - y) + z)`, not `(x - (y + z))`
- Most operators also have a binding strength or precedence.
  E.g.: `(x + y * z)` is `(x + (y * z))`, not `((x + y) * z)`
- Resolving precedence by hand is a lot of fun, but a bit out of scope for CSC 3F002 EP, at least at present
  - Look up: *Pratt parsing* aka *precedence climbing*
  - Also see: *shunting yard algorithm*
  - May revisit later if the need arises
- For now: we will use the built-in precedence declaration of PLY/Yacc.

Operator Precedence in PLY/Yacc Parser

Still in `parser.py`:

```
# To declare precedence, there needs to be a ‘precedence’ table (tuple)
# that is listed in order of increasing precedence
precedence = (
  (’left’, ’PLUS’, ’MINUS’),                # left-assoc., low precedence
  (’left’, ’TIMES’, ’DIV’, ’MODULUS’),  # left-assoc., medium precedence
  (’right’, ’UMINUS’),                      # right-assoc., high precedence
)

# ...

parser = yacc.yacc(start=’expr’)
```

```
>>> from scanner import lexer
>>> from parser import parser
>>> parser.parse(’x + y * z’, lexer=lexer)
<...> # BinopApp(’PLUS’, Variable(’x’), BinopApp(’TIMES’, Variable(’y’), Variable(’z’)))
>>> parser.parse(’(x + y) * z’, lexer=lexer)
<...> # BinopApp(’TIMES’, BinopApp(’PLUS’, Variable(’x’), Variable(’y’)), Variable(’z’))
>>>
```

Sequences (and Separators)

- Core BNF has no built in support for sequences, but:
- To get a sequence of $\langle\text{item}\rangle$s, one can do:
  - Zero or more $\langle\text{item}\rangle$s as an auxiliary nonterminal $\langle\text{items}\rangle$

$$\langle\text{items}\rangle ::= \epsilon \mid \langle\text{item}\rangle\ \langle\text{items}\rangle$$

  - One or more $\langle\text{item}\rangle$s as an auxiliary nonterminal $\langle\text{items1}\rangle$

$$\langle\text{items1}\rangle ::= \langle\text{item}\rangle\ \langle\text{items}\rangle$$

  - Easy to modify to separate individual $\langle\text{item}\rangle$s by `COMMA`, say:

$$\langle\text{items-comma}\rangle ::= \epsilon \mid \langle\text{items-comma1}\rangle$$
$$\langle\text{items-comma1}\rangle ::= \langle\text{item}\rangle \mid \langle\text{item}\rangle\ \texttt{COMMA}\ \langle\text{items-comma1}\rangle$$

Extended BNF

- Instead of tokens names (e.g. `COMMA`), use their textual form (e.g. `","`)
- Group items with parentheses – note: the parentheses are not part of the language being described
- Indicate zero-or-more repetitions with *
  One-or-more repetitions with $^+$
  Zero-or-one occurrences with $^?$

$\langle\text{program}\rangle ::= \texttt{"def main() \{"}\ \langle\text{vardecl}\rangle^*\ \langle\text{stmt}\rangle^*\ \texttt{"\}"}$

$\langle\text{procdef}\rangle ::= \texttt{"def"}\ \texttt{IDENT}\ \texttt{"("}\ (\langle\text{param}\rangle\ (\texttt{","}\ \langle\text{param}\rangle)^*)^?\ \texttt{")"}\ \ldots$ ($\sim$ lab 4)

Step 3: Syntactic Analysis

- Grammars specified in BNF are context free, so:
  - Cannot detect if every variable has been declared
  - Cannot detect if no variable has multiple declarations
  - Similarly, cannot detect that every procedure has a unique definition
- The output of the parser is sometimes called a parse tree rather than an AST
- Syntactic checks are implemented by traversing the parse tree.
- For the body of a procedure:
  - Whenever a ⟨vardecl⟩ is encountered, check that:
    1. the variable has not already been declared
    2. the initializer is a well formed expression
  - Whenever an ⟨expr⟩ession is encountered, check that:
    1. Every variable has been declared earlier
  - More complex checks as the language grows...

Lab 1: Preview

- You will write:
  - A lexer and parser for a fragment of BX using PLY
- Fragment of BX:
  - Will be a single function (`main`) program, straightline code, and **int**eger vars

$$\langle\text{program}\rangle ::= \texttt{"def main() \{"}\ \langle\text{stmt}\rangle^*\ \texttt{"\}"}$$

$$\langle\text{stmt}\rangle ::= \langle\text{vardecl}\rangle \mid \langle\text{assign}\rangle \mid \langle\text{print}\rangle$$

$$\langle\text{vardecl}\rangle ::= \texttt{"var"}\ \texttt{IDENT}\ \texttt{"="}\ \langle\text{expr}\rangle\ \texttt{": int;"}$$