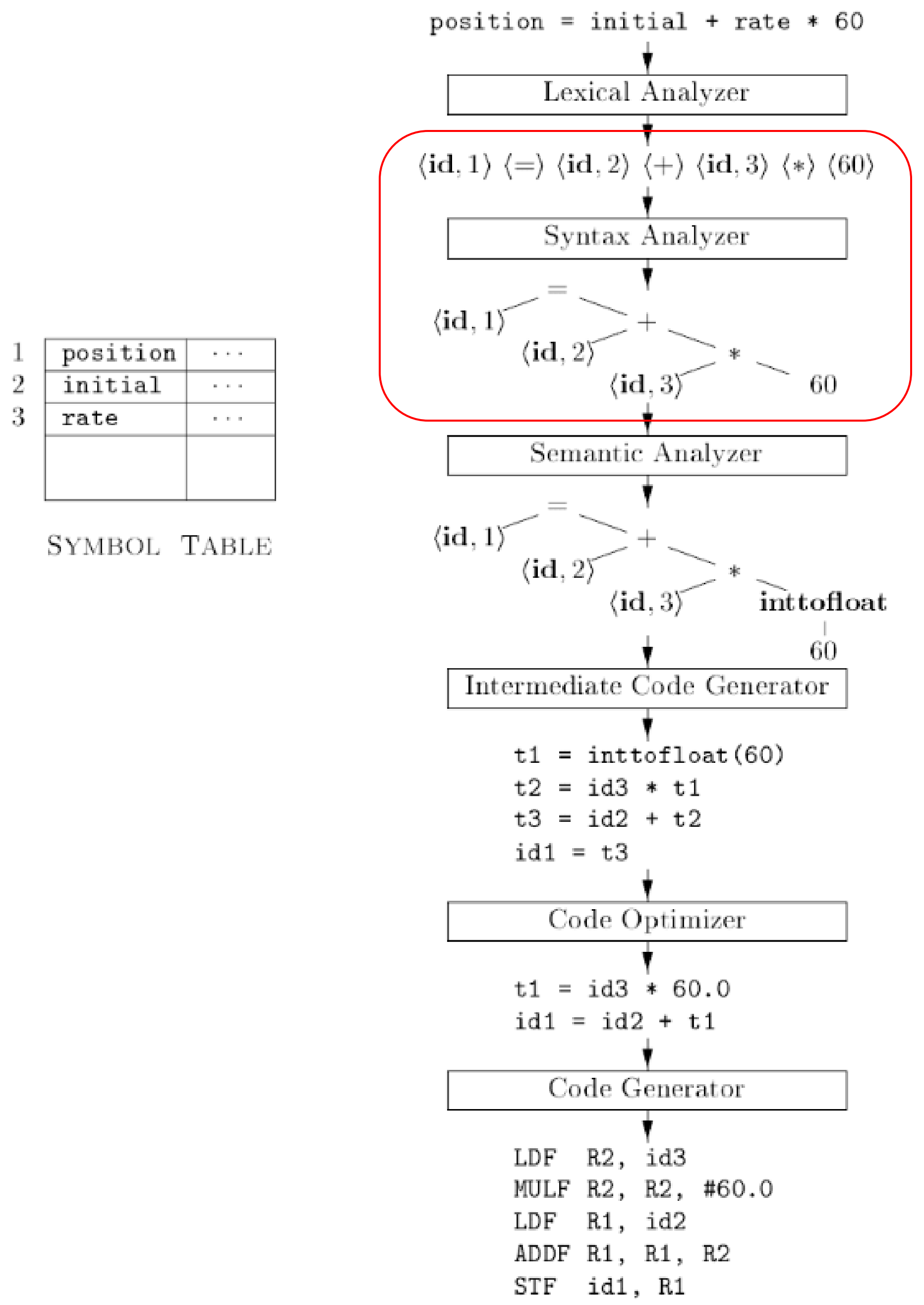
Top-Down Parsing

COP-3402 Systems Software

Paul Gazzillo



1	position	···
2	initial	···
3	rate	···

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

```
t1 = id3 * 60.0
id1 = id2 + t1
```

```
LDF  R2, id3
MULF R2, R2, #60.0
LDF  R1, id2
ADDF R1, R1, R2
STF  id1, R1
```

Figure 1.7: Translation of an assignment statement

Core Idea: Construct Syntax Tree

- Grammar *describes* the syntax
- Recognizer *matches* string against grammar
- Parser *constructs* parse tree or syntax tree from string
- Parser is recognizer plus tree construction

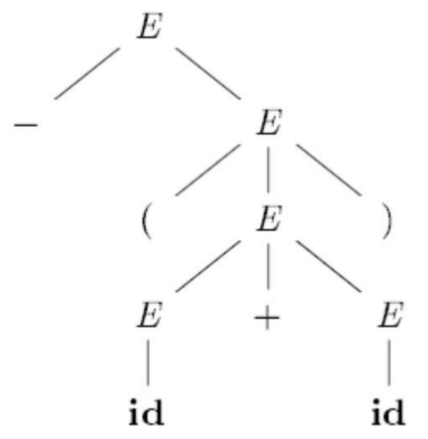


Figure 4.3: Parse tree for $-(\mathbf{id} + \mathbf{id})$

Parsers are Either Top-Down or Bottom-Up

- Top-down
  - Start from root of parse tree
  - Construct tree nodes until leaves (tokens)
  - Bottom-up
- Bottom-up
  - Start from leaves (tokens)
  - Build tree nodes until root

Top-Down Parsing Algorithms

- Recursive descent
    - Convert grammar to recursive functions
- Table-driven top-down parsing
    - Construct parsing table
    - (will not talk about in this class)

Recursive Descent

- Each nonterminal is a function
- Each terminal matches an input character
- Each production forms the body of nonterminal functions
- What's wrong with this algorithm?

```
E -> E + T | T
T -> T * F | F
F -> 0 | 1
```

Eliminating Left Recursion

- Algorithms to do this automatically
  - Left recursion elimination
  - Left factoring
- Result

```
E -> E + T | T
T -> T * F | F
F -> 0 | 1
```

```
E  -> TE'
E' -> +TE'| epsilon
T  -> FT'
T' -> *FT' | epsilon
F  -> 0 | 1
```

Backtracking vs. Predictive Parsing

- Backtracking
  - Search for matching grammar productions
  - Return when failed
  - Try next production
  - Expensive
- Predictive parsing
  - Determine production using terminal
  - Linear time
  - Easier to write
  - Limited grammars

Writing a Recursive Descent Parser

Demo

PL/0 Parser Overview