Abstract Syntax Trees

COP-3402 Systems Software

Paul Gazzillo

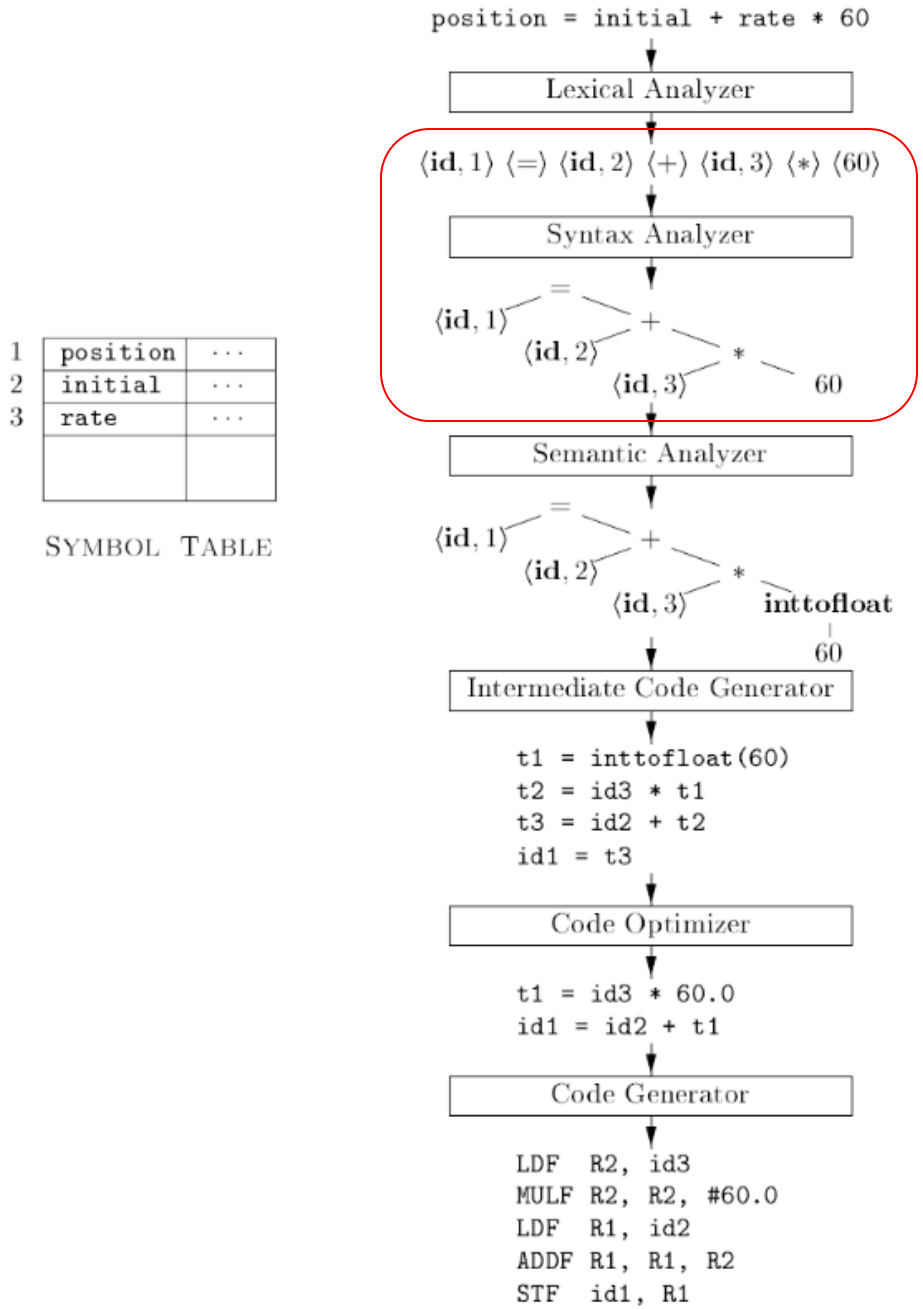

Figure 1.7: Translation of an assignment statement

Core Idea: Construct *Abstract* Syntax Tree

- Grammar *describes* the syntax
- Recognizer *matches* string against grammar
- Parser *constructs* abstract syntax tree (AST) for the given string

AST Specification

ast.yml

Example parser output

Parse Tree vs AST

- Parse tree
  - Each node is a symbol from the grammar
- AST
  - Removes elements unnecessary for compilation
  - Uses convenient data structures: lists, unions; not just tree
    - Easier development for later compiler phases
  - With OOP, use class definitions for each tree node

Example Parse Tree and AST

"1+0*v"

Why Use an AST? Cleaner Design

- Punctuation and keywords are removed
  - More compact
  - Memory efficiency
  - No need to program around useless tree nodes
- Better data structures
  - Choose the right data structure and the algorithm naturally follows
- Simpler expression trees
  - No need to preserve all nonterminals
  - Code generation is simpler
- Ultimately about design
  - Critical for large software systems
  - What's the right API for each component?

Syntax-Directed Definition of AST Creation

- Annotate grammar with *semantic actions*
  - Rules define translation, evaluation, etc
- Each nonterminal has a value
  - e.g., a pointer to an AST node
- Each production's action updates values
  - e.g., create a new AST node
- Semantic actions happen during parsing
  - Inside of each recursive function

```
block   ::== vardecls funcdecls statement
   block.value = newBlock();
   block.value.vardelcs = vardecls.value;
   block.value.funcdecls = funcdecls.value;
   block.value.statement = statement.value

term  ::== factor1 [ factorop factor2 ]
   if factor2.value exists
     term.value = newBinaryExpression()
     term.value.binary_left = factor1.value
     term.value.binary_right = factor2.value
   else
     term.value = newUnaryExpression()
     term.value.unary_expression = factor1.value
```

Advantages of Our Example

- Fewer nodes
  - 5 in AST vs 9 in parse tree
- Fewer, more descriptive nodes
  - BinaryExpression, UnaryExpression vs expr, simpleexpr, term
- Simpler semantics
  - BinaryExpression encompasses all operations between two expressions
  - Factors are typed, e.g., NumberFactor, VariableFactor:
    - Easier implementation of later phases

PL/0 Parser

- Parser does two things simultaneously
  - Recursive descent implementation of the grammar
  - AST generation during parser
- Each nonterminal's function
  - Calls other nonterminal functions and matches tokens
  - creates and returns an AST node

PL/0 Parser Overview

Constructing ASTs