CS 314 Principles of Programming Languages

Lecture 10: Syntax Directed Translation

Zheng (Eddy) Zhang

Rutgers University

February 19, 2018

Class Information

- Homework 2 is still being graded.
- Project 1 and homework 4 will be released tomorrow.

Review: Recursive Descent Parsing

Recursive descent parser for LL(1)

- Each **non-terminal** has an associated parsing procedure that can recognize any sequence of tokens generated by that **non-terminal**
- There is a main routine to initialize all globals (e.g:the ***token*** variable in previous code example) and call the start symbol. On return, check whether ***token*** == EOF, and whether errors occurred.
- Within a parsing procedure, both **non-terminals** and **terminals** are matched:

  ➡ Non-terminal A: call procedure for A

  ➡ Token t: compare t with current the first of the remaining tokens; If matched, **consume input**, otherwise, ERROR

- Parsing procedure may contain code that performs some useful "computations" (*syntax directed translation)*

Review: Syntax Directed Translation

Examples:

- Interpreter
- Code generator
- Type checker
- Performance estimator

Use hand-written recursive descent LL(1) parser

Example: the Original Parser

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	Rule 1	Rule 2	
< digit >		Rule 3	

```
void expr( ):  // return value of the expression
        int val1, val2; // two values
        switch token {
                case +:       token := next_token( );
                              expr( );
                case 0..9:    digit( );
                …
        }

void digit( ):  // return value of constant
        switch token {
                case 1: token := next_token( );
                case 2: token := next_token( );
               …
        }
```

Example: Interpreter

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	Rule 1	Rule 2	
< digit >		Rule 3	

Interpreter

```
int expr( ) {
   int val1, val2; // two values
   switch token {
      case +:    token := next_token( );
                 val1 =  expr( ) ;
                 val2 = expr( );
                 return val1 + val2;
      case 0..9:
                 return digit( );
      …
   } // End switch case
} //End expr()

int digit( ):  // return value of constant
   switch token {
      case 1: token := next_token( ); return 1;
      case 2: token := next_token( ); return 2;
      …
   } // End switch case
}// End digit( )
```

Original

```
void expr( ) {

   switch token {
      case +:    token := next_token( );
                 expr( ) ;
                 expr( );

      case 0..9:
                 digit( );
      …
   } // End switch case
} //End expr()

void digit( ):  // return value of constant
   switch token {
      case 1: token := next_token( );
      case 2: token := next_token( );
      …
   } // End switch case
}// End digit( )
```

Example: Interpreter

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	Rule 1	Rule 2	
< digit >		Rule 3	

Interpreter

```
int expr( ) {
   int val1, val2; // two values
   switch token {
      case +:    token := next_token( );
                 val1 =  expr( ) ;
                 val2 = expr( );
                 return val1 + val2;
      case 0..9:
                    return digit( );
      …
   } // End switch case
} //End expr()

int digit( ):  // return value of constant
   switch token {
      case 1: token := next_token( ); return 1;
      case 2: token := next_token( ); return 2;
      …
   } // End switch case
}// End digit( )
```

What happens when parsing expression
“ + 2 + 1 2”

The parsing produces
5

Example: Simple Code Generator

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	Rule 1	Rule 2	
< digit >		Rule 3	

Code Generator

```
int expr( ) {
   int target_reg; // target register
   int reg1, reg2; // source registers
   switch token {
      case +:
         token := next_token( );
         target_reg = next_register( );
         reg1 =  expr( ) ;
         reg2 = expr( );
         print_inst(ADD, reg1, reg2, target_reg);
         return target_reg;
      case 0..9:
         return digit( );
      …
   } // End switch case
} //End expr()
```

```
int digit( ):  // return value of constant
   int target_reg;  // target register
   switch token {
      case 1:
         token := next_token( );
         target_reg = next_register( );
         print_inst(LOADI, 1, target_reg);
         return target_reg;
      case 2:
         ….
   } // End switch case
}// End digit( )
```

Example: Simple Code Generator

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	Rule 1	Rule 2	
< digit >		Rule 3	

Code Generator

```
int expr( ) {
  int target_reg; // target register
  int reg1, reg2; // source registers
  switch token {
    case +:
      token := next_token( );
      target_reg = next_register( );
      reg1 =  expr( ) ;
      reg2 = expr( );
      print_inst(ADD, reg1, reg2, target_reg);
      return target_reg;
    case 0..9:
      return digit( );
    …
  } // End switch case
} //End expr()
```

```
int digit( ):  // return value of constant
  int target_reg;  // target register
  switch token {
    case 1:
      token := next_token( );
      target_reg = next_register( );
      print_inst(LOADI, 1, target_reg);
      return target_reg;
    case 2:
      ….
  } // End switch case
}// End digit( )
```

“ADD r<reg1>, r<reg2> => r<target_reg>”

Example: Simple Code Generator

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	Rule 1	Rule 2	
< digit >		Rule 3	

Code Generator

```
int expr( ) {
  int target_reg; // target register
  int reg1, reg2; // source registers
  switch token {
    case +:
      token := next_token( );
      target_reg = next_register( );
      reg1 =  expr( ) ;
      reg2 = expr( );
      print_inst(ADD, reg1, reg2, target_reg);
      return target_reg;
    case 0..9:
      return digit( );
    …
  } // End switch case
} //End expr()
```

```
int digit( ):  // return value of constant
  int target_reg;  // target register
  switch token {
    case 1:
      token := next_token( );
      target_reg = next_register( );
      print_inst(LOADI, 1, target_reg);
      return target_reg;
    case 2:
      ….
  } // End switch case
}// End digit( )
```

“ LOADI 1 => r<target_reg>”

Example: Code Generator (cont.)

What happens when you parse the program: " + **2** + **1 2**" ?

First call to next_register() will return 1

```
int expr( ) {
   int target_reg; // target register
   int reg1, reg2; // source registers
   switch token {
      case +:
         token := next_token( );
         target_reg = next_register( );
         reg1 =  expr( ) ;
         reg2 =  expr( );
         print_inst(ADD, reg1, reg2, target_reg);
         return target_reg;
      case 0..9:
         return digit( );
      …
   } // End switch case
} //End expr()
```

The parsing produces:

```
LOADI  2 => r2
LOADI  1 => r4
LOADI  2 => r5
ADD  r4, r5 => r3
ADD  r2, r3 => r1
```

```
int digit( ):  // return value of constant
   int target_reg;  // target register
   switch token {
      case 1:
         token := next_token( );
         target_reg = next_register( );
         print_inst(LOADI, 1, target_reg);
         return target_reg;
      case 2:
         ….
   } // End switch case
}// End digit( )
```

Example: Type Checker

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	Rule 1	Rule 2	
< digit >		Rule 3	

```
string expr( ) { // returns type expression
  string  type1, type2; // other type expressions
  switch token {
      case +:
        token := next_token( );
        type1 = expr( );
        type2 = expr( );
        if (type1 == "int" && type2 == "int"){
          return "int";
        } else{
          return "error";
        }
      case 0..9:
        return digit ( );
        …
  }
}
```

```
string digit( ) {  // returns type expression
  switch token {
    case 1:    token := next_token( );
               return "int";
    case 2:    token := next_token( );
               return "int";
                …
                }
                }
```

Example: Type Checker (cont.)

What happens when you parse subprogram

“ + **2** + **1 2**” ?

The parsing produces:

“int”

Example: Basic Performance Predictor

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	Rule 1	Rule 2	
< digit >		Rule 3	

```
int expr( ) {  // returns cycles
  int cyc1, cyc2; // subexpression cycles
  switch token {
    case +:
      token := next_token ( )
      cyc1 = expr( );
      cyc2 = expr( );
      // ADD takes 2 cycles
      return cyc1 + cyc2 + 2;
    case 0..9:
      return digit ( );
    …
  }
}
```

```
int digit( ) {  // returns cycles
  switch token {
    case 1:
      token := next_token ( );
      return 1; // LOADI takes 1 cycle
    case 2:
      token := next_token ( );
      return 1; // LOADI takes 1 cycle
      …
  }
}
```

Example: Basic Performance Predictor (cont.)

What happens when you parse subprogram
“ + **2** + **1 2**” ?

ADD takes 2 cycles

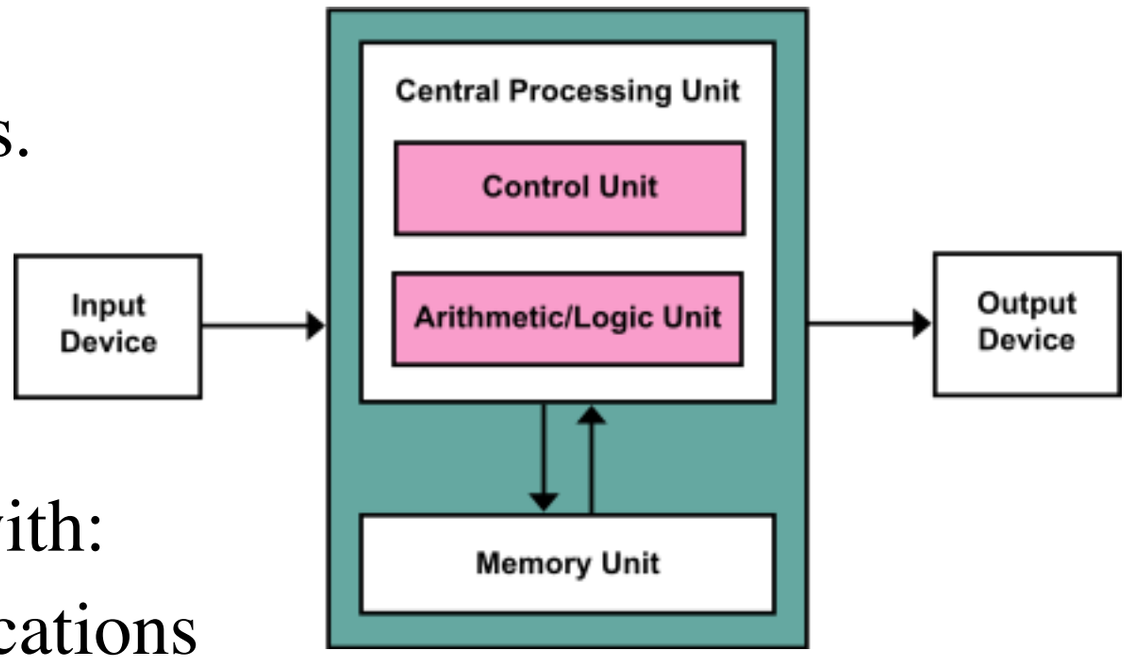
LOADI takes 1 cycle

The parsing produces:

7

Imperative Programming Language

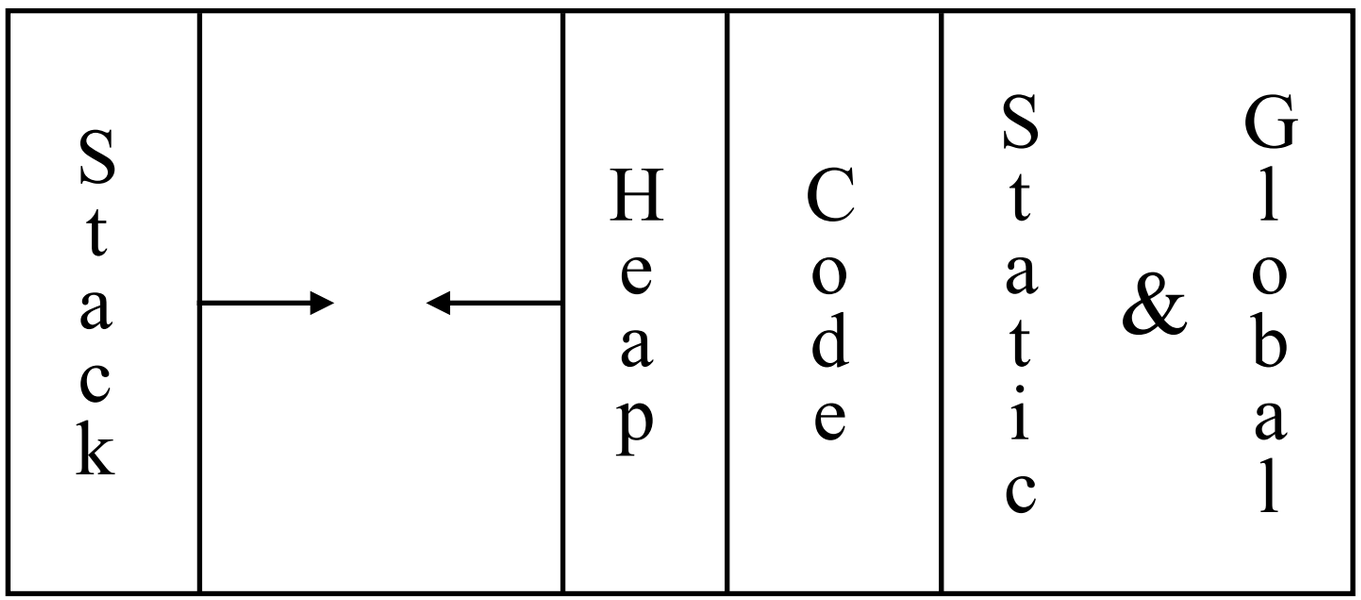
Imperative:
Sequence of state-changing actions.



- Manipulate an abstract machine with:
  1. Variables naming memory locations
  2. Arithmetic and logical operations
  3. Reference, evaluate, assign operations
  4. Explicit control flow statements
- Key operations: Assignment and control flow
- Fits the Von Neumann architecture closely

Run-time Storage Organization

Typical memory layout



Most Language runtime layout the address space in a similar way

- Pieces (stack, heap, code & globals) may move, but all will be there
- Stack and heap grow toward each other
- Arrays live on one of the stacks, in the global area, or in the heap

Stack vs Heap

Stack:

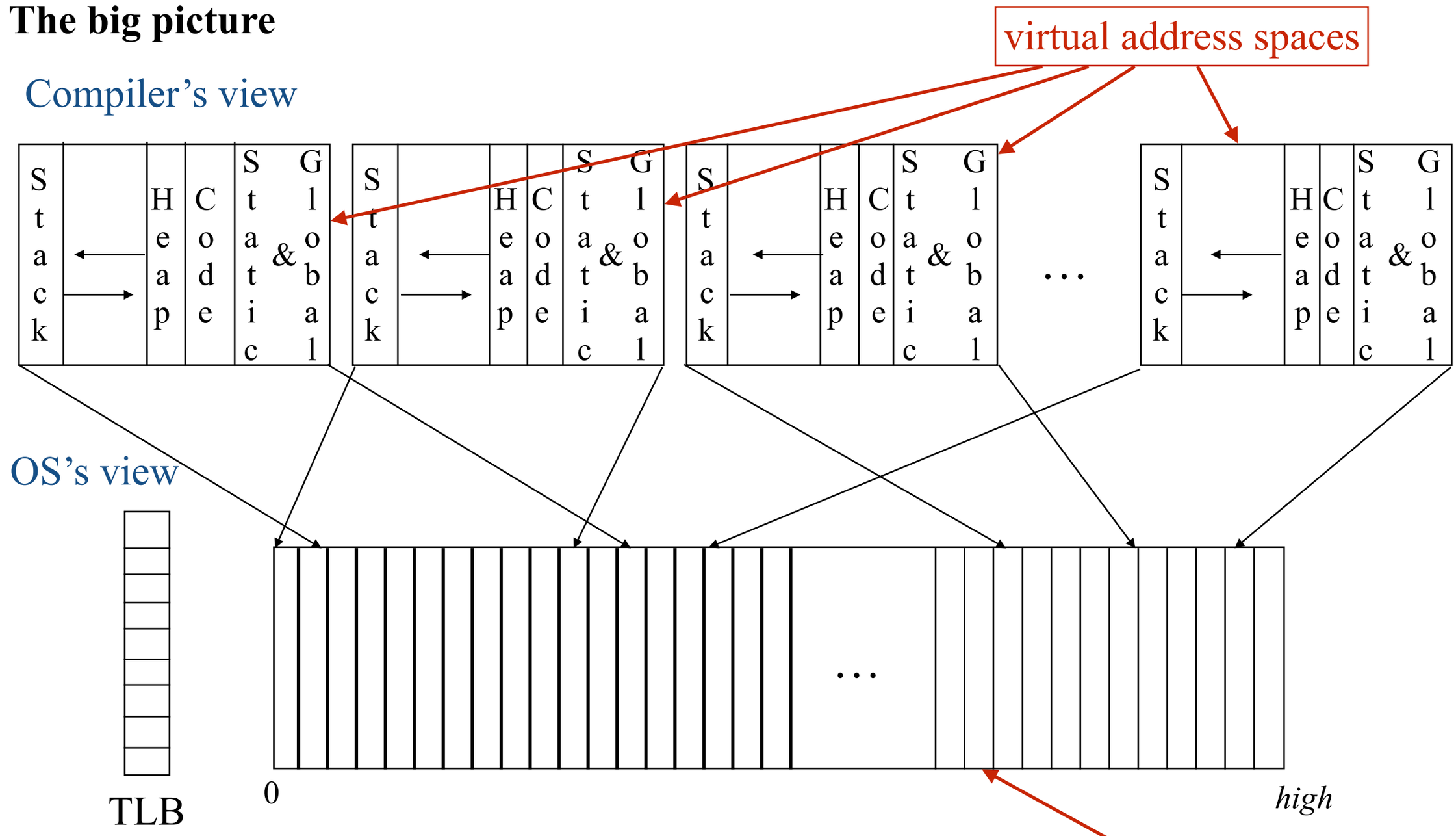
- Procedure activations, statically allocated local variables, parameter values
- Lifetime same as subroutine in which variables are declared
- Stack frame is pushed with each invocation of a subroutine, and popped after subroutine exit

Heap:

- Dynamically allocated data structure, whose size may not be known in advance
- Lifetime extends beyond subroutine in which they are created
- Must be explicitly freed or garbage collected

How Does Address Space Mapping Work?

The big picture



TLB is an address cache used by the **OS** to speed virtual-to-physical address translation. A process may have >1 level **TLB**

Next Lecture

Things to do:

- Start programming in C.
- Read Scott, Chapter 8.1 - 8.2; ALSU 7.1 - 7.3.
- Next time:
  - Procedure abstractions; Runtime stack; Scoping