LLVM Tutorial

Front-end Compilation

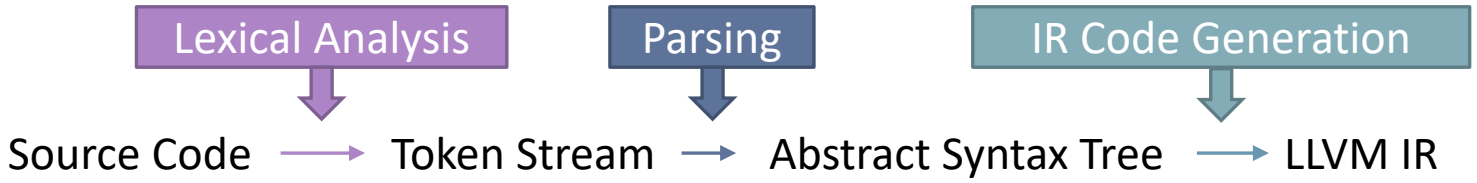
2019. 04. 11

Front-end Compilation

- Translate source code into **LLVM IR** code

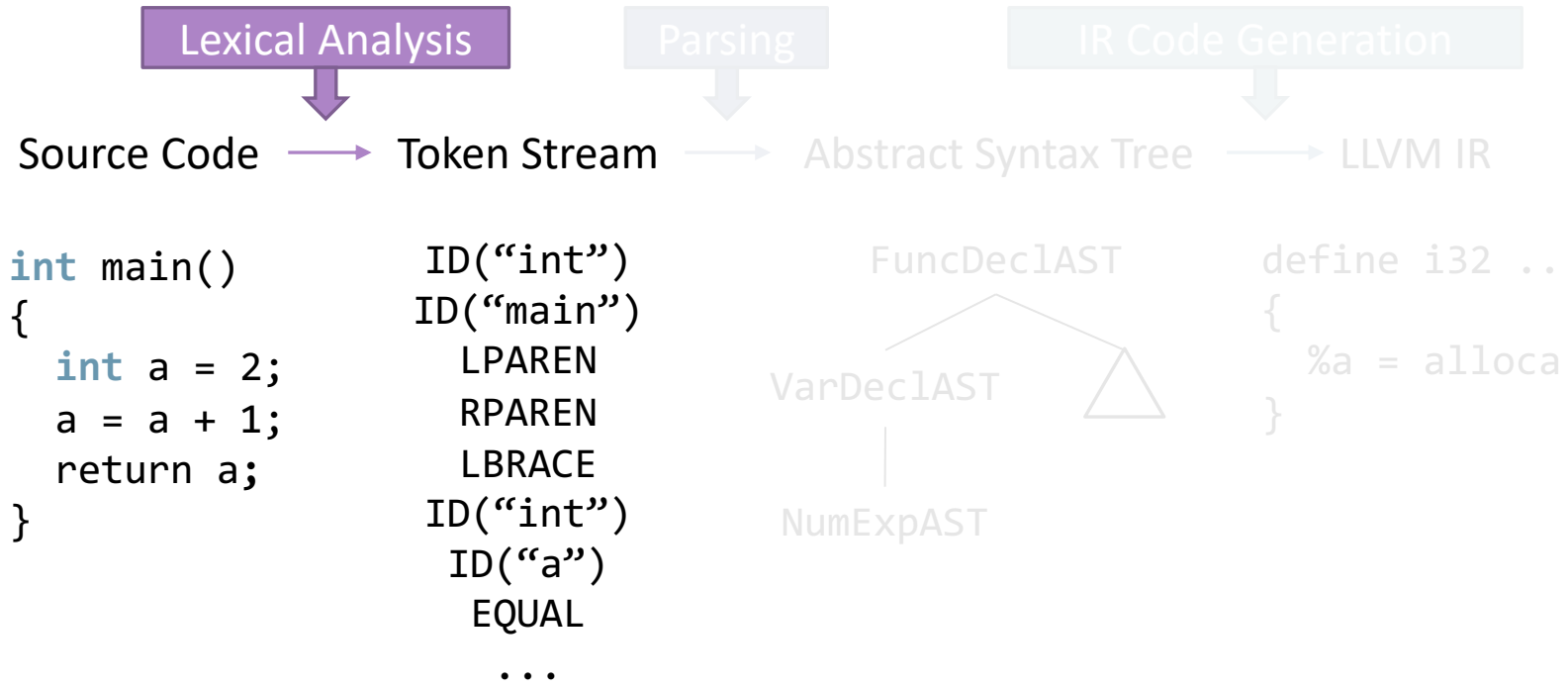
- Less dependent on LLVM
  - Need to use little LLVM core libraries
  - Various ways to implement front-end compiler (Ocaml, …)

- Overall Process



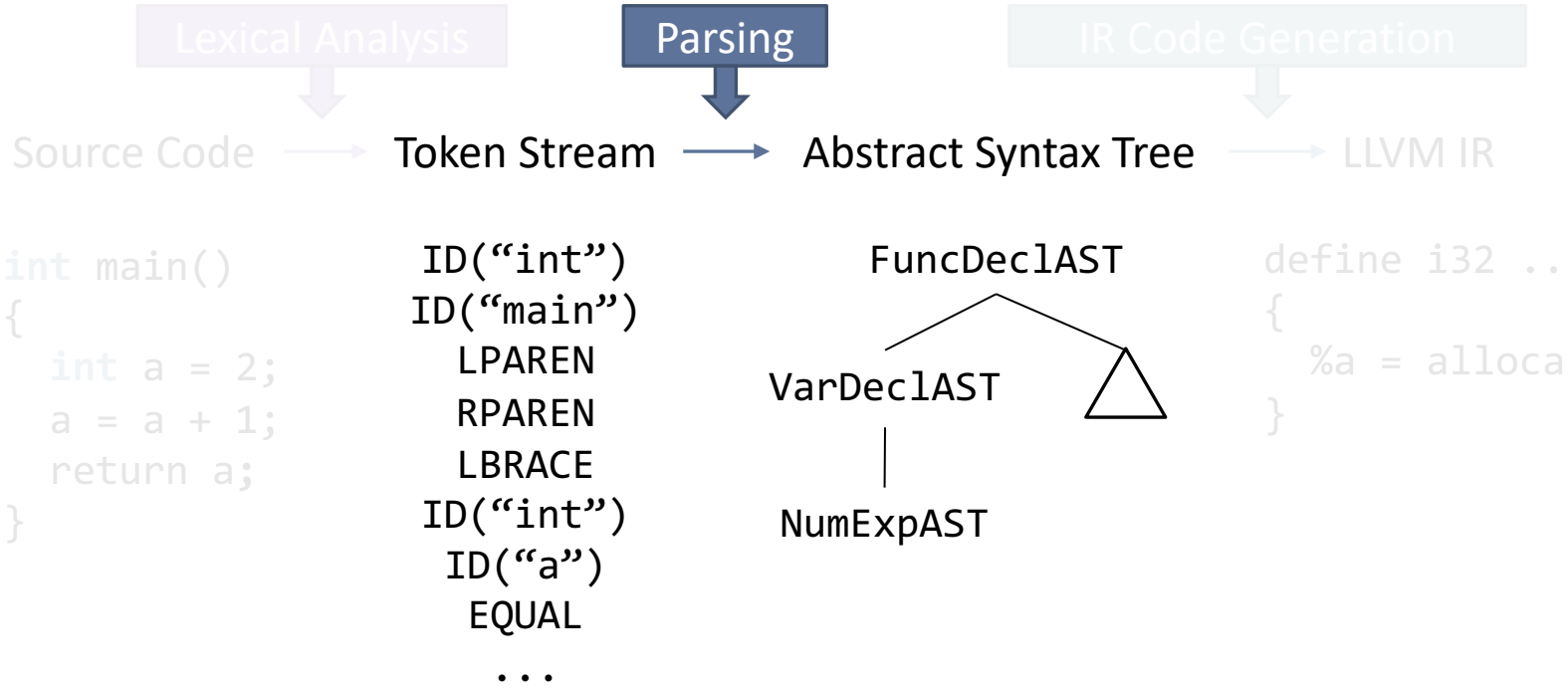
Front-end Compilation

- Example in C language



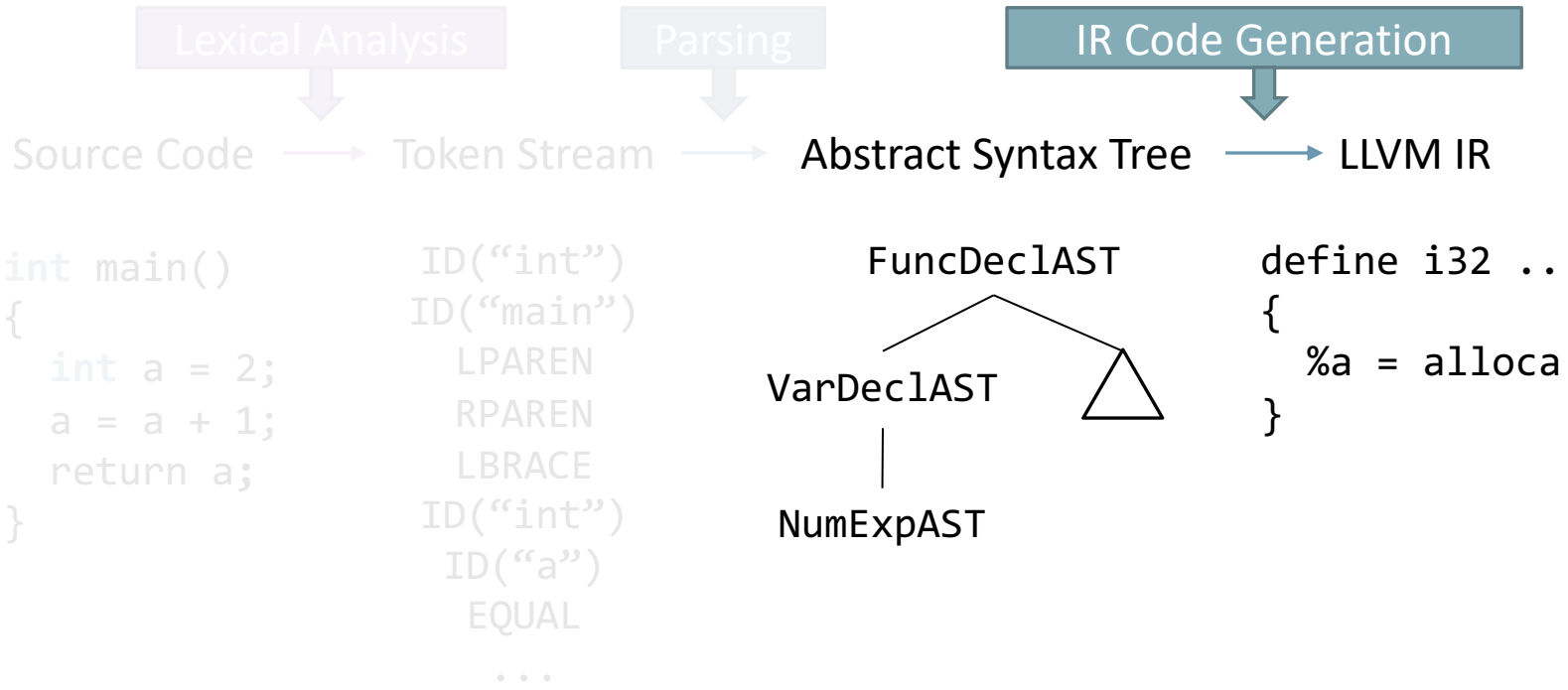
Front-end Compilation

- Example in C language



Front-end Compilation

- Example in C language



Toy Language

- Definition in a Context-free Grammar
  - $Fun \rightarrow \textbf{fun id}\ (\ Args\ )\{\ Exp\ \}$
  - $Args \rightarrow Args\ ,\ Args$
  - $Args \rightarrow \textbf{id}$

Non-terminal

  - $Exp \rightarrow Exp\ BinOp\ Exp$
  - $Exp \rightarrow \textbf{id}$
  - $Exp \rightarrow \textbf{num}$

Terminal

  - $BinOp\ \rightarrow\ +\ |\ -$

* **id** (Identifier): a string word starting with an alphabet
* **num** (Number): an integer

Fun Language

- Example Programs
  - Function that increments *num* by one

```
fun inc (num) {
  num + 1
}
```

  - Function that adds two integers

```
fun add (a, b) {
  a + b
}
```

Lexical Analysis

- Tokenize a stream of input in a program

Input

Program

```
fun inc (num) {
  num + 1
}
```

Output

Token Stream

```
FUN, ID("inc"),
LPAREN, ID("num"), RPAREN,
LBRACE, ID("num"), PLUS,
NUM(1), RBRACE
```

- Lexer
  - A compiler module that performs lexical analysis

How to Design Lexer

1) Define token types

Definition

$Fun \rightarrow \textbf{fun id}\ (\ Args\ )\{\ Exp\ \}$

$Args \rightarrow Args\ ,\ Args$

$Args \rightarrow \textbf{id}$

$Exp \rightarrow Exp\ BinOp\ Exp$

$Exp \rightarrow \textbf{id}$

$Exp \rightarrow \textbf{num}$

$BinOp \rightarrow\ +\ |\ -$

Token Types

```
FUN
ID(string)
NUM(integer)
LPAREN
RPAREN
LBRACE
RBRACE
COMMA
PLUS
MINUS
```

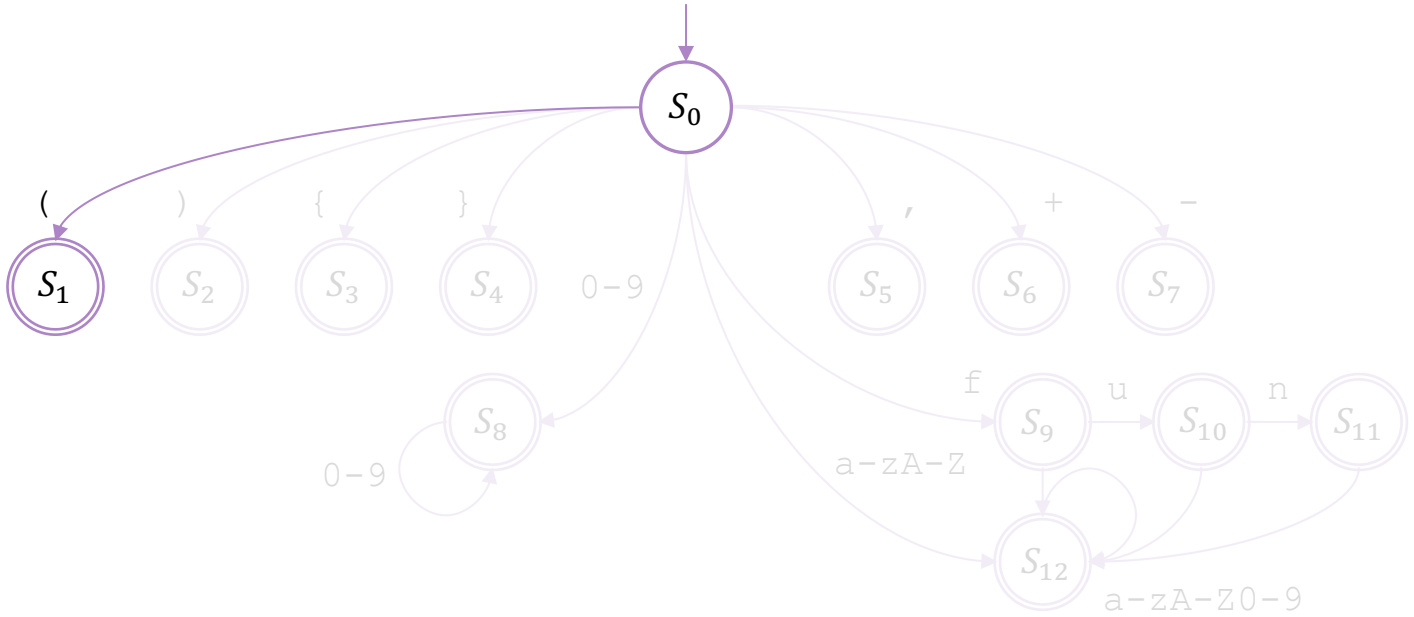
How to Design Lexer

2) Draw **a finite state machine** for lexical analysis
   - Express each token as a regular expression
   - Convert the regular expression into a finite state machine

   - Leaving edges are uniquely labeled
     - Skip white spaces
     - Emit a token at final states
     - Return to the initial state after emitting a token

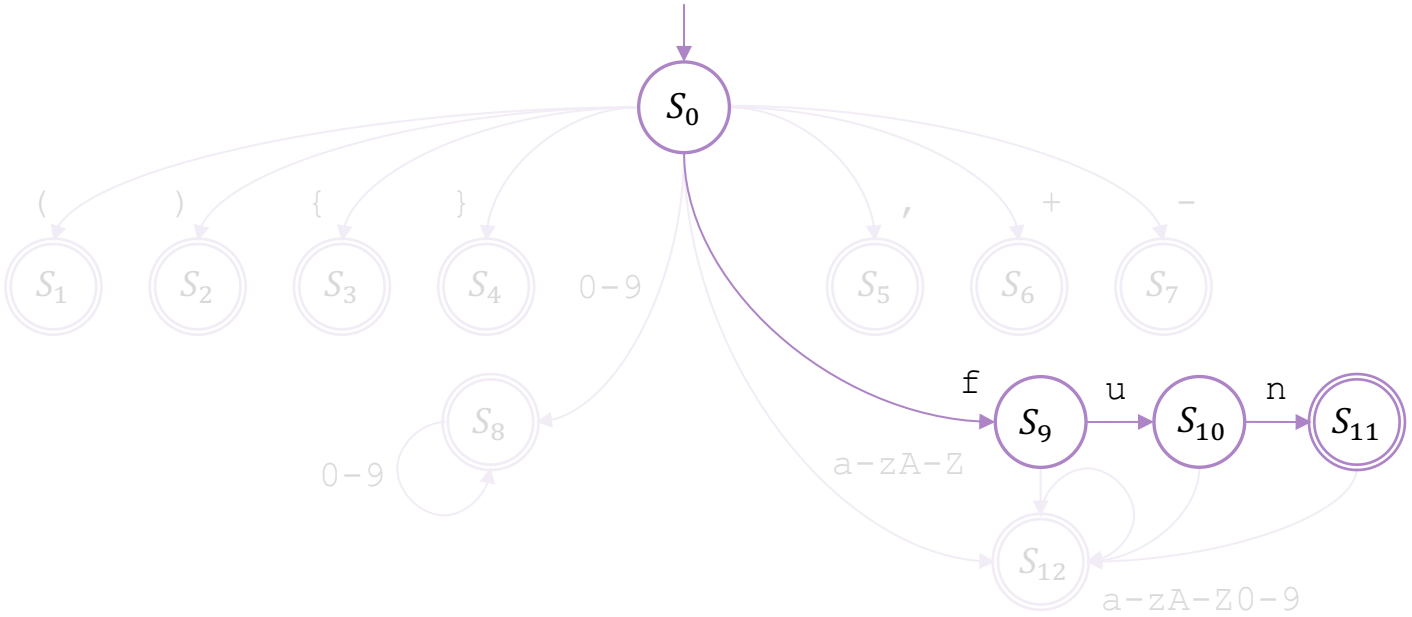
How to Design Lexer

2) Draw **a finite state machine** for lexical analysis
   - Token Type: LPAREN = (



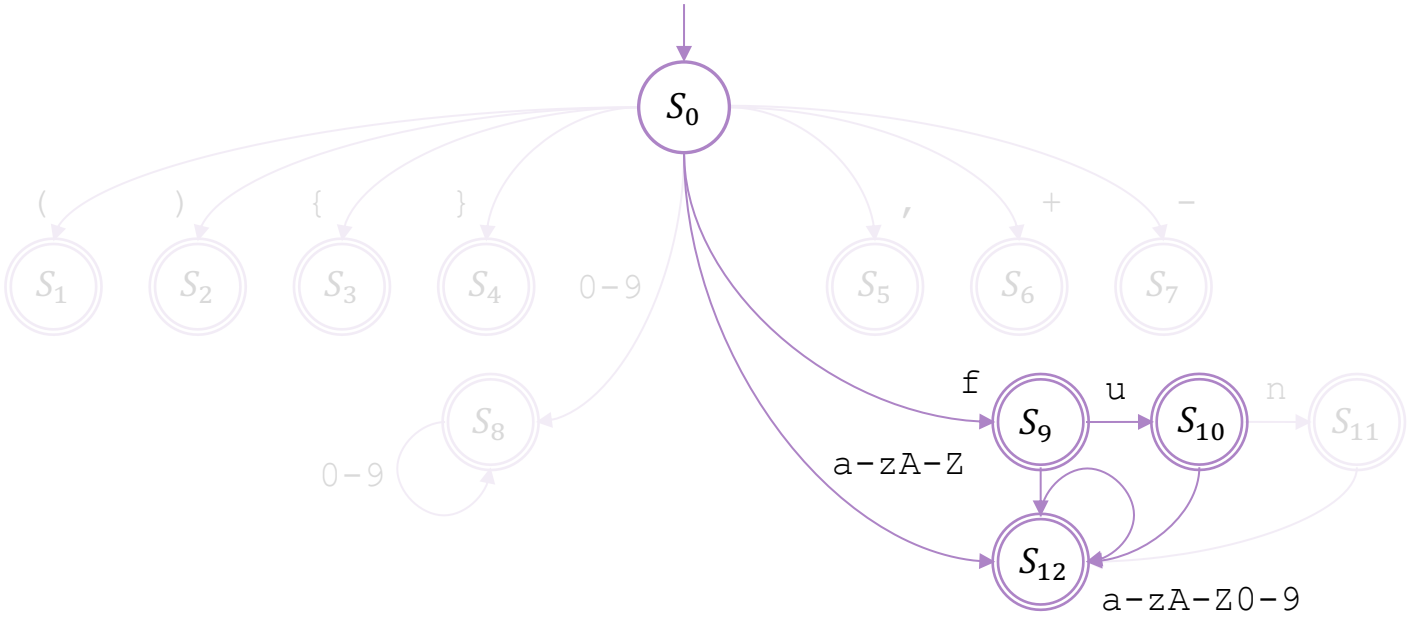
How to Design Lexer

2) Draw **a finite state machine** for lexical analysis
   - Token Type: FUN = `fun`



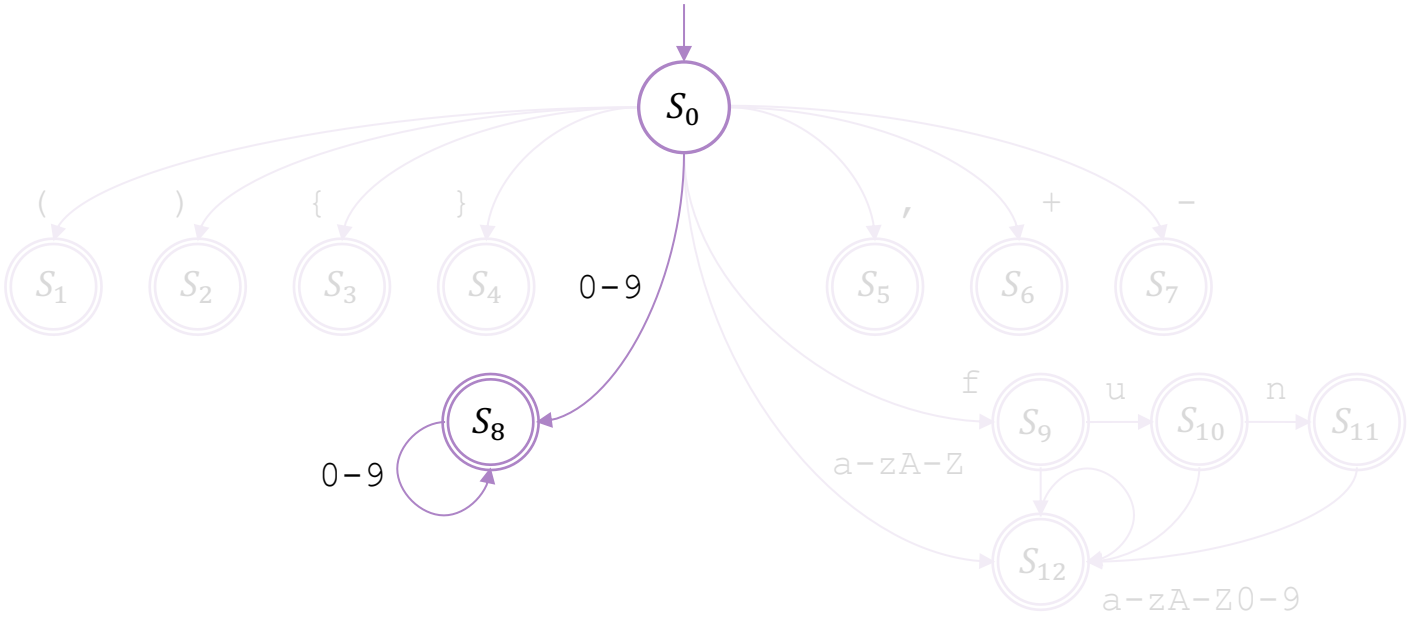
How to Design Lexer

2) Draw **a finite state machine** for lexical analysis
   - Token Type: `ID(string) = [a-zA-Z][a-zA-Z0-9]*`



How to Design Lexer

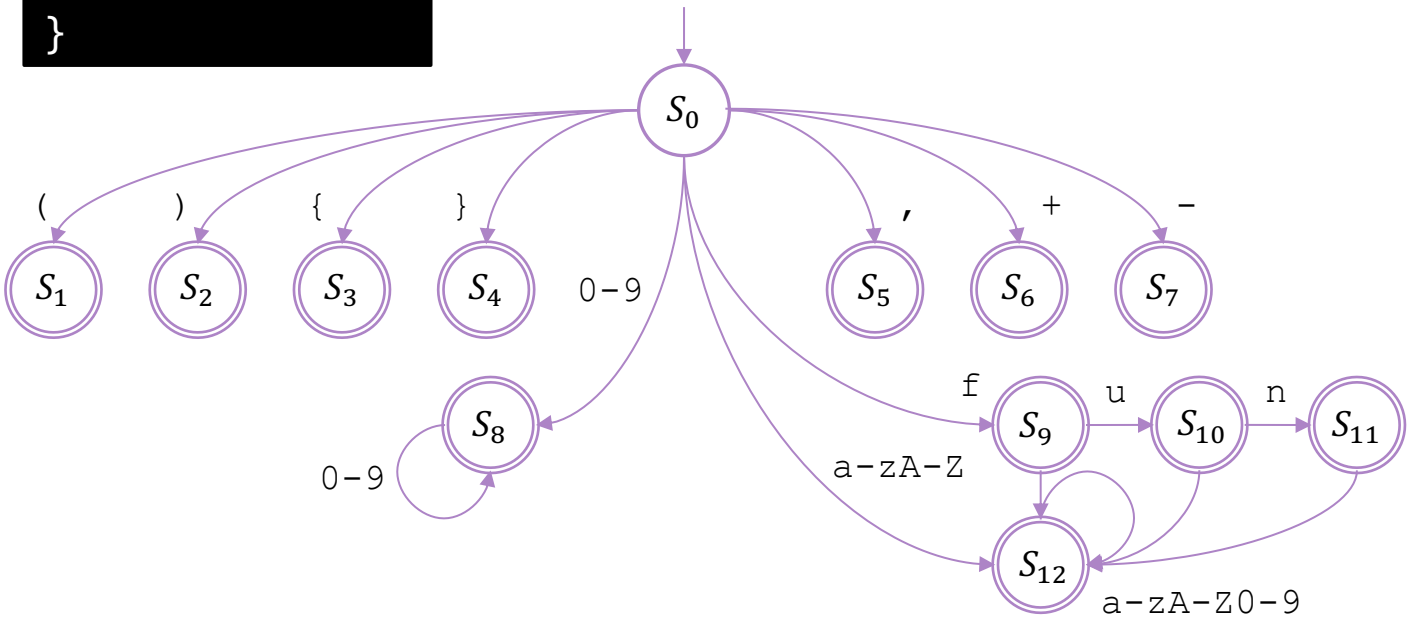
2) Draw **a finite state machine** for lexical analysis
   - Token Type: `NUM(int) = [0-9]+`



How to Design Lexer

2) Draw **a finite state machine** for lexical analysis

```
fun inc (num) {
  num + 1
}
```



How to Implement Lexer

- Define token types

```
// The lexer returns tokens [0-255] if it is an unknown character,
// otherwise one of these for known things.
enum Token {
  tok_eof = -1,

  tok_fun = -2,

  tok_id = -3,
  tok_num = -4
};
```

- Containers for tokens

```
static std::string IdStr; // Filled in if tok_id
static double NumVal;     // Filled in if tok_num
```

How to Implement Lexer

- Main Function
  - consume() returns a character read from input file stream

```
/// gettok - Return the next token from input file stream.
static int gettok() {
  static int LastChar = ' ';

  // Skip any whitespace.
  while (isspace(LastChar))
    LastChar = consume();
```

How to Implement Lexer

- Main Function
  - `isalpha(char)` returns whether the character is an alphabet or not

```
if (isalpha(LastChar)) { // id: [a-zA-Z][a-zA-Z0-9]*
  IdStr = LastChar;
  while (isalnum((LastChar = consume())))
    IdStr += LastChar;

  if (IdStr == "fun")
    return tok_fun;

  return tok_id;
}
```

How to Implement Lexer

- Main Function
  - `isdigit(char)` returns whether the character is a digit or not

```
if (isdigit(LastChar)) { // Number: [0-9]+
  string NumStr;
  do {
    NumStr += LastChar;
    LastChar = consume();
  } while (isdigit(LastChar));

  NumVal = stoi(NumStr);
  return tok_num;
}
```

How to Implement Lexer

- Main Function

```
  // Check for end of file.  Don't eat the EOF.
  if (LastChar == EOF)
    return tok_eof;

  // Otherwise, just return the character as its ascii value.
  int ThisChar = LastChar;
  LastChar = consume();
  return ThisChar;
}
```

Practice 1: Extend the Syntax

- Goal
  - Extend the toy language with If-Else expression
    - $Exp \rightarrow \textbf{if}\ (\ Exp\ )\ Exp\ \textbf{else}\ \ Exp\ \textbf{end}$

- Steps
  1) Define token types for
     - IF, ELSE, END
  2) Emit the tokens in `gettok` function
  3) Extend printtok function to print the tokens
  4) Test the lexer with if_end.toy

Parsing

- Analyze the (syntactic) structure of a program
- Generate an abstract syntax tree for a token stream

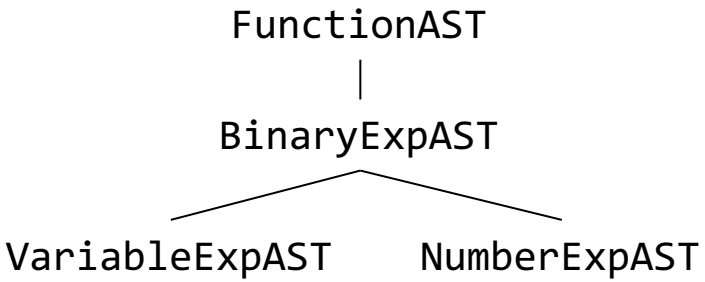
Input

Token Stream

```
FUN, ID(“inc”),
LPAREN, ID(“num”), RPAREN,
LBRACE, ID(“num”), PLUS,
NUM(1), RBRACE
```

Output

Abstract Syntax Tree (AST)



Concrete Parse Tree

- Derivation tree of a program
  - Each **internal node** is labeled with a non-terminal
  - Each **leaf node** is labeled with a terminal

$Fun \rightarrow \textbf{fun id} \; (\; Args \;)\{\; Exp \;\}$

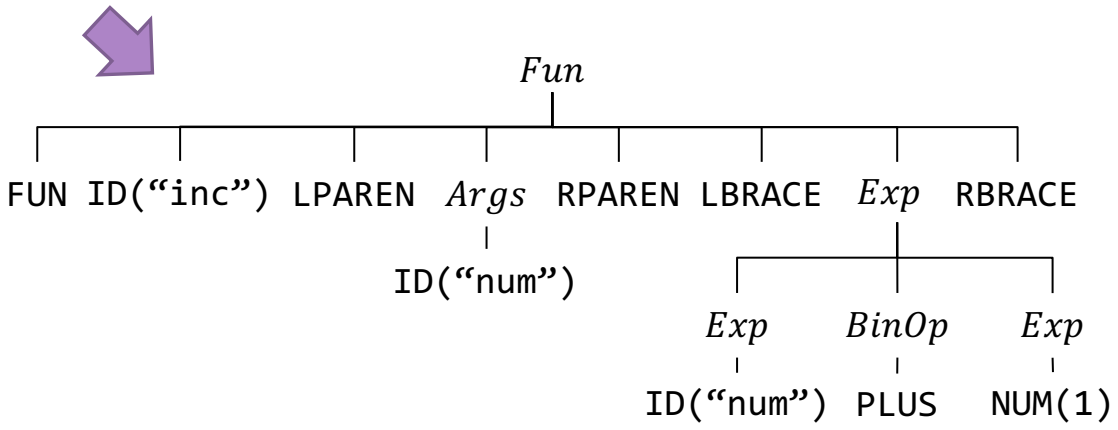
$Args \rightarrow \textbf{id}$

$Exp \rightarrow Exp \; BinOp \; Exp$

$Exp \rightarrow \textbf{id}$

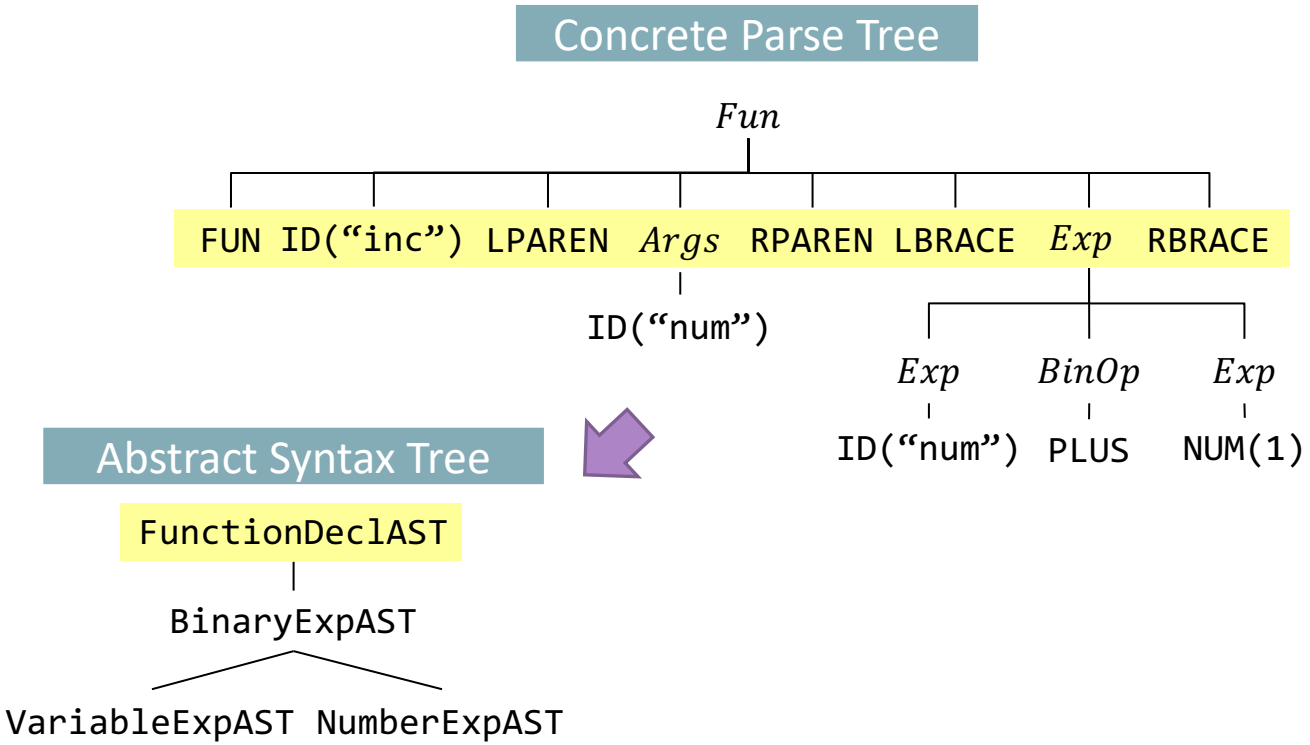
$Exp \rightarrow \textbf{num}$

```
fun inc (num) {
  num + 1
}
```



Abstract Syntax Tree

- Similar to concrete parse tree, except redundant tokens left out



How to Design Parser

- LL(1) Parsing
  - Top-down parsing = Predictive parsing
  - Simple parsing, Low parsing power
  - Non-terminal + Current token → Which derivation rule to use?
    - Example parse rule
      - Non-terminal: S (Start Symbol)
      - Next token: FUN
      - Derivation rule: $Fun \rightarrow \mathbf{fun\ id}\ (\ Arg\ )\{\ Exp\ \}$
  - A **parsing table** contains parsing rules
    - Parsing table ~ Parser

How to Design Parser

- LL(1) Parsing
  - Compute three values to generate a parsing table
    - $Nullable$
      - γ is nullable if γ can be derived to empty string
    - $First(\gamma)$
      - A set of all terminal symbols that can begin any string derived from γ
      - $Fun \rightarrow \textbf{fun id}\ (\ Args\ )\{\ Exp\ \}$, then $\textbf{fun} \in First(Fun)$
    - $Follow(X)$
      - A set of all terminal symbols that can immediately follow X in a derivation
      - $Fun \rightarrow \textbf{fun id}\ (\ Args\ )\{\ Exp\ \}$, then $) \in Follow(Args)$

How to Design Parser

- LL(1) Parsing
  - Calculate $(Nullable)$, $First(\gamma)$, $Follow(X)$

$Fun \rightarrow$ **fun** **id** ($Args$){ Exp }

$Args \rightarrow Args$, $Args$

$Args \rightarrow$ **id**

$Exp \rightarrow Exp$ ***BinOp*** Exp

$Exp \rightarrow$ **id**

$Exp \rightarrow$ **num**

$BinOp \rightarrow + \mid -$

	First	*Follow*
Function	**fun**	
Args	**id**	,)
Exp	**id num**	} + −

How to Design Parser

- LL(1) Parsing
  - Generate a parsing table

	First	*Follow*
Fun	**fun**	
$Args$	**id**	,)
Exp	**id num**	} + −

$Fun \rightarrow \textbf{fun id} \; (\; Args \;)\{\; Exp \;\}$

$Args \rightarrow Args \;,\; Args$

$Args \rightarrow \textbf{id}$

$Exp \rightarrow Exp \; BinOp \; Exp$

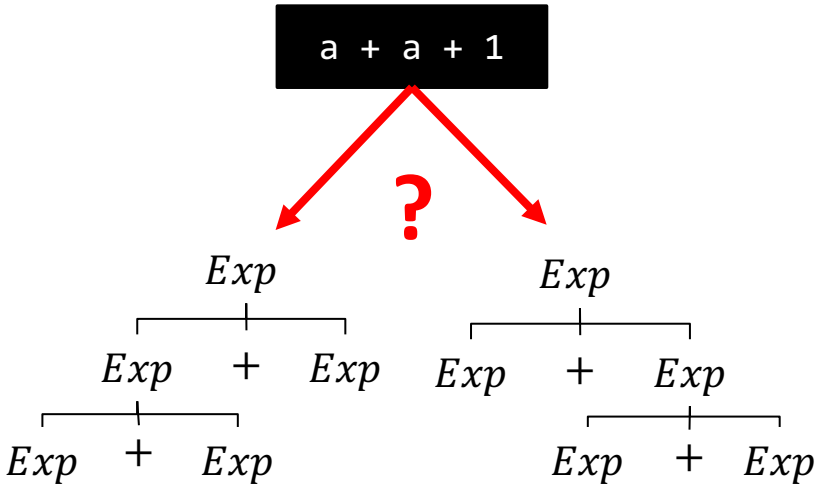
$Exp \rightarrow \textbf{id}$

$Exp \rightarrow \textbf{num}$

	fun	**id**	**num**
Fun	$Fun \rightarrow \textbf{fun id} \; (\; Args \;)\{\; Exp \;\}$		
$Args$		$Args \rightarrow Args \;,\; Args$ $Args \rightarrow \textbf{id}$	
Exp		$Exp \rightarrow Exp \; BinOp \; Exp$ $Exp \rightarrow \textbf{id}$	$Exp \rightarrow Exp \; BinOp \; Exp$ $Exp \rightarrow \textbf{num}$

How to Design Parser

- Ambiguous grammar
  - Multiple derivation trees for the same code



How to Design Parser

- LL(1) Parsing
  - **Left-recursive** grammar cannot be LL(1)
  - Change to right-recursive using the following rule

$$\begin{array}{l} X \to X\gamma \\ X \to \alpha \end{array} \longrightarrow \begin{array}{l} X \to \alpha X' \\ X' \to \epsilon \\ X' \to \gamma X' \end{array}$$

$$Args \to Args\ ,\ Args$$
$$Args \to \mathbf{id}$$

$$Exp \to Exp\ BinOp\ Exp$$
$$Exp \to \mathbf{id}$$
$$Exp \to \mathbf{num}$$

How to Design Parser

- LL(1) Parsing
  - **Left-recursive** grammar cannot be LL(1)
  - Change to right-recursive using the following rule

$$X \to X\gamma \quad X \to \alpha \;\Rightarrow\; X \to \alpha X' \quad X' \to \epsilon \quad X' \to \gamma X'$$

$Args \to Args\ ,\ Args$

$Args \to \mathbf{id}$

$Args \to \mathbf{id}\ Args'$

$Args' \to \epsilon$

$Args' \to ,\ Args\ Args'$

$Exp \to Exp\ BinOp\ Exp$

$Exp \to \mathbf{id}$

$Exp \to \mathbf{num}$

$\Rightarrow$

$Exp \to \mathbf{id}\ Exp'$

$Exp \to \mathbf{num}\ Exp'$

$Exp' \to \epsilon$

$Exp' \to BinOp\ Exp\ Exp'$

	Nullable	*First*	*Follow*
Fun	x	**fun**	
Args	x	**id**	)
Args′	o	**,**	)
Exp	x	**id** **num**	}
Exp′	o	+ -	}

How to Design Parser

- LL(1) Parsing
  - Re-generate the parsing table

Current Token
ex) ID

	fun	id	num
Fun	$Fun \rightarrow \textbf{fun id} \; (\; Args \;) \{ \; Exp \; \}$		
$Args$		$Args \rightarrow \textbf{id} \; Args'$	
$Args'$			
Exp		$Exp \rightarrow \textbf{id} \; Exp'$	$Exp \rightarrow \textbf{num} \; Exp'$
Exp'			

Current Parsing function
ex) `ParseExpression`

How to Design Parser

- LL(1) Parsing
  - Re-generate the parsing table

,	)	+/-	}
Fun			
$Args$			
$Args'$ $Args' \rightarrow ,Args\ Args'$	$Args' \rightarrow \epsilon$		
Exp			
Exp'		$Exp' \rightarrow BinOp\ Exp\ Exp'$	$Exp' \rightarrow \epsilon$

How to Design Parser

- Define Abstract Syntax Trees

Definition

$Fun \rightarrow \textbf{fun id} \; (\; Args \;)\{\; Exp \;\}$

$Args \rightarrow Args \;,\; Args$

$Args \rightarrow \textbf{id}$

$Exp \rightarrow Exp \; BinOp \; Exp$

$Exp \rightarrow \textbf{id}$

$Exp \rightarrow \textbf{num}$

$BinOp \rightarrow \; + \mid -$

Abstract Syntax Tree

```
FunctionDeclAST
ExpAST
BinaryExpAST
VariableExpST
NumberExpST
```

How to Implement Parser

- Define Abstract Syntax Trees
  - `FunctionDeclAST`

$Fun \rightarrow \textbf{fun id}\ (\ Args\ )\{\ Exp\ \}$

```
class FunctionDeclAST {
  string FunName;
  vector<string> ArgNames;

  unique_ptr<ExpAST> Body;

public:
  FunctionDeclAST(string FunName, vector<string> &ArgNames,
                  unique_ptr<ExpAST> Body)
    : FunName(FunName), ArgNames(ArgNames), Body(move(Body)) {}
};
```

How to Implement Parser

- Define Abstract Syntax Trees
  - `BinaryExpAST`

$Exp \rightarrow Exp\ BinOp\ Exp$

```
class BinaryExpAST : public ExpAST {
  char Op;
  unique_ptr<ExpAST> LHS, RHS;

public:
  BinaryExpAST(char op, unique_ptr<ExpAST> LHS,
               unique_ptr<ExpAST> RHS)
    : Op(op), LHS(move(LHS)), RHS(move(RHS)) {}
};
```

How to Implement Parser

- Define Abstract Syntax Trees
  - `VariableExpAST`

```
class VariableExpAST : public ExpAST {
  string Name;

public:
  VariableExpAST(const string &Name) : Name(Name) {}
};
```

  - `NumberExpAST`

```
class VariableExpAST : public ExpAST {
  string Name;

public:
  VariableExpAST(const string &Name) : Name(Name) {}
};
```

How to Implement Parser

- Main Loop
  - Top-level parsing
    - Program = A set of function definitions

```
static void MainLoop() {
  while (1) {
    switch (CurTok) {
      case tok_eof:
        return;
      case tok_fun:
        HandleFunctionDefinition();
        break;
      default:
        // Skip token for error recovery.
        getNextToken();
    }
  }
}
```

How to Implement Parser

- Implement parsing functions for each AST
  - FunctionDeclAST

$Fun \rightarrow \textbf{fun id} \; (\; Args \;) \{ \; Exp \; \}$

```
static unique_ptr<FunctionDeclAST> ParseFunctionDefinition() {
  getNextToken(); // eat fun.
  if (CurTok != tok_id) {
    cerr << "Expected function name in function definition" << endl;
    return nullptr;
  }
  string FunName = IdStr;

  getNextToken();
  if (CurTok != '(') {
    cerr << "Expected '(' in function definition" << endl;
    return nullptr;
  }
```

How to Implement Parser

- Implement parsing functions for each AST
  - FunctionDeclAST

$Fun \rightarrow \textbf{fun id} \; (\; Args \;) \{ \; Exp \; \}$
$Args \rightarrow \textbf{id} \; Args'$
$Args' \rightarrow \epsilon$
$Args' \rightarrow , Args \; Args'$

```
vector<string> ArgNames;

getNextToken();
if (CurTok != tok_id) {
  cerr << "Expected argument name in function definition" << endl;
  return nullptr;
}
ArgNames.push_back(IdStr);
```

How to Implement Parser

- Implement parsing functions for each AST
  - FunctionDeclAST

$Fun \rightarrow \textbf{fun id} \; (\; Args \;) \{ \; Exp \; \}$

```
getNextToken();
while (CurTok != ')') {
  if (CurTok != ',') {
    cerr << "Expected ',' in function definition" << endl;
    return nullptr;
  }

  getNextToken();
  if (CurTok != tok_id) {
    cerr << "Expected argument name in function definition" << endl;
    return nullptr;
  }
  ArgNames.push_back(IdStr);

  getNextToken();
}
```

How to Implement Parser

- Implement parsing functions for each AST
  - FunctionDeclAST

$Fun \rightarrow \textbf{fun id}\ (\ Args\ )\{\ Exp\ \}$

```
getNextToken();
if (CurTok != '{') {
  cerr << "Expected '{' in function definition" << endl;
  return nullptr;
}

if (auto E = ParseExpression())
  return llvm::make_unique<FunctionDeclAST>(FunName, ArgNames, move(E));

return nullptr;
```

How to Implement Parser

- Implement parsing functions for each AST
  - ExpAST

```
static unique_ptr<ExpAST> ParseExpression() {
  std::unique_ptr<ExpAST> LHS;

  getNextToken();
  switch (CurTok) {
    case tok_id:
      LHS = llvm::make_unique<VariableExpAST>(IdStr);
      break;
    case tok_num:
      LHS = llvm::make_unique<NumberExpAST>(NumVal);
      break;
    default:
      cerr << "Unknown token when expecting an expression" << endl;
      return nullptr;
  }

  return ParseExpression(move(LHS));
}
```

	id	num
Exp	$Exp \rightarrow \mathbf{id}\ Exp'$	$Exp \rightarrow \mathbf{num}\ Exp'$

How to Implement Parser

- Implement parsing functions for each AST
  - ExpAST

```
static unique_ptr<ExpAST> ParseExpression(unique_ptr<ExpAST> LHS) {
  char Op;
  getNextToken();

  switch (CurTok) {
    case '+':
      Op = '+';
      break;
    case '-':
      Op = '-';
      break;
    case '}':
      return LHS;
    default:
      cerr << "Unknown token when expecting '+', '-', or '}'" << endl;
      return nullptr;
  }

  if (auto RHS = ParseExpression())
    return llvm::make_unique<BinaryExpAST>(Op, move(LHS), move(RHS));

  return nullptr;
}
```

	+/-	}
Exp'	$Exp' \rightarrow BinOp\ Exp\ Exp'$	$Exp' \rightarrow \epsilon$

Practice 2: Extend the Syntax

- Goal
  - Extend the toy language with If-Else expression
    - $Exp \rightarrow \textbf{if}\ (\ Exp\ )\ Exp\ \textbf{else}\ \ Exp\ \textbf{end}$

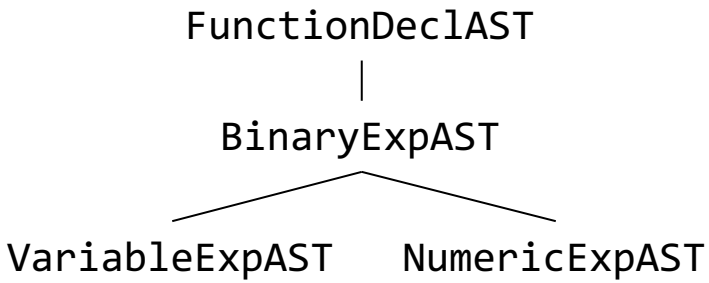
- Steps
  1) Extend the parsing table for If-Else expression
     - Which token requires the derivation rule?
  2) Define IfElseAST for If-Else expression
  3) Implement the parsing function for IfElseAST

IR Code Generation

- Generate intermediate representation code from AST

Input

Abstract Syntax Tree (AST)



Output

LLVM IR

```
define i32 ..
{
  %a = alloca
}
```

How to Implement CodeGen

- Declare necessary variables

```
static LLVMContext Context;
static IRBuilder<> Builder(Context);
static std::unique_ptr<Module> M;
static std::map<std::string, Value *> VariableMap;
```

- Create a module object

```
M = llvm::make_unique<Module>("Toy", Context);
```

How to Implement CodeGen

- Add Codegen functions for each AST

```
class FunctionDeclAST {
  string FunName;
  vector<string> ArgNames;

  unique_ptr<ExpAST> Body;

public:
  FunctionDeclAST(string FunName, vector<string> &ArgNames,
                  unique_ptr<ExpAST> Body)
    : FunName(FunName), ArgNames(ArgNames), Body(move(Body)) {}
  virtual Function *Codegen();
};
```

How to Implement CodeGen

- Add Codegen functions for each AST

```
Function *FunctionDeclAST::Codegen() {
  // Make the function type:  double(double,double) etc.
  std::vector<Type*>ArgTypes(ArgNames.size(),Type::getInt32Ty(Context));
  FunctionType *FunType = FunctionType::get(
      Type::getInt32Ty(Context), ArgTypes, false);

  Function *F = Function::Create(FunType, Function::ExternalLinkage,
      FunName, M.get());

  // Set names for all arguments.
  unsigned Idx = 0;
  for (auto &Arg : F->args()) {
    Arg.setName(ArgNames[Idx]);
    VariableMap[ArgNames[Idx]] = &Arg;
    Idx++;
  }
```

How to Implement CodeGen

- Add Codegen functions for each AST

```
  BasicBlock *BB = BasicBlock::Create(Context, "entry", F);
  Builder.SetInsertPoint(BB);
  if (Value *RetVal = Body->Codegen()) {
    // Finish off the function.
    Builder.CreateRet(RetVal);

    // Validate the generated code, checking for consistency.
    verifyFunction(*F);

    return F;
  }

  F->eraseFromParent();
  return nullptr;
}
```

Call Codegen for AST

Practice 3: Extend the Syntax

- Goal
  - Extend the toy language with If-Else expression
    - $Exp \rightarrow \textbf{if}\ (\ Exp\ )\ Exp\ \textbf{else}\ \ Exp\ \textbf{end}$

- Steps
  1) Implement the Codegen function for IfElseAST
  2) Test with a sample code

Backup Slides

LL(1) Parsing

- Before making a parser, we need to compute 3 values
  - $Nullable$
    - For each γ corresponding to RHS of production, γ is nullable if γ can be derived to empty string (ϵ)
  - $First(\gamma)$
    - For each γ corresponding to RHS of production, $First(\gamma)$ is a set of all terminal symbols that can begin any string derived from γ
    - Ex: $S \rightarrow if\ E\ then\ S\ else\ S$, $if \in First(S)$
  - $Follow(X)$
    - For each non-terminal X in grammar, $Follow(X)$ is a set of all terminal symbols that can immediately follow X in a derivation
    - Ex: $S \rightarrow if\ E\ then\ S\ else\ S, then \in Follow(E)$

LL(1) Parsing

- How to compute $Nullable$
  - If $X \rightarrow \epsilon$, X is nullable
  - X is nullable if every symbol $S \in X$ is nullable

- Example

$Z \rightarrow XYZ$ $\quad$ $Y \rightarrow c$ $\quad$ $X \rightarrow a$

$Z \rightarrow d$ $\quad$ $Y \rightarrow \epsilon$ $\quad$ $X \rightarrow bYe$

	Initial	Iteration 1	Iteration 2
X	No	No	No
Y	No	Yes	Yes
Z	No	No	No

LL(1) Parsing

- How to compute $First(X)$
  - If X is a terminal symbol, then $First(X) = \{X\}$
  - If X is a non-terminal and $X \rightarrow Y_1 Y_2 Y_3 \ldots Y_n$, then
    - $First(Y_1) \in First(X)$
    - $First(Y_2) \in First(X)$ if Y_1 is nullable
    - $First(Y_3) \in First(X)$ if Y_1 and Y_2 are nullable
    - $First(Y_n) \in First(X)$ if $Y_1, Y_2, \ldots, Y_{n-1}$ are nullable

LL(1) Parsing

- How to compute $Follow(X)$
  - X,Y: Non-terminals
  - γ,γ_1,γ_2: Strings of terminals and non-terminals
  - If grammar includes production: $X \rightarrow \gamma Y$
    - $Follow(X) \in Follow(Y)$
  - If grammar includes production: $X \rightarrow \gamma_1 Y \gamma_2$
    - $First(\gamma_2) \in Follow(Y)$
    - $Follow(X) \in follow(Y)$, if γ_2 is nullable

LL(1) Parsing

- Example

$$Z \to XYZ \qquad Y \to c \qquad X \to a$$
$$Z \to d \qquad Y \to \epsilon \qquad X \to bYe$$

Initial

	nullable	first	follow
X	No		
Y	No		
Z	No		

Iteration 1

	nullable	first	follow
X	No	a,b	
Y	Yes	c	
Z	No	d	

Iteration 2

	nullable	first	follow
X	No	a,b	
Y	Yes	c	
Z	No	d,a,b	

Iteration 2

	nullable	first	follow
X	No	a,b	c,d,a,b
Y	Yes	c	e,d,a,b
Z	No	d,a,b	

LL(1) Parsing

- Predictive Parsing Table
  - Enter $S \rightarrow \gamma$ in row S, column T: for each $T \in First\ (\gamma)$
  - If γ is nullable, enter $S \rightarrow \gamma$ in row S, column T: for each $T \in follow(S)$
  - Entry in row S, column T tells parser which clause to execute if current function is $S()$ and next token is T
  - Blank entries are syntax errors

LL(1) Parsing

- Example

$Z \to XYZ \quad Y \to c \quad X \to a$
$Z \to d \quad Y \to \epsilon \quad X \to bYe$

	nullable	first	follow
X	No	a,b	c,d,a,b
Y	Yes	c	e,d,a,b
Z	No	d,a,b	

- Build predictive parsing table from nullable, first, and follow sets

	a	b	c	d	e
X	$X \to a$	$X \to bYe$			
Y	$Y \to \epsilon$	$Y \to \epsilon$	$Y \to c$	$Y \to \epsilon$	$Y \to \epsilon$
Z	$Z \to XYZ$	$Z \to XYZ$		$Z \to d$	