University of Kansas | Drew Davidson

EECS 665 COMPILER CONSTRUCTION

Scope

Last Time

Lecture Review - LR Parsing

LR Parser Construction

- LR Parsers
- Building SLR Parser tables

<u>You Should Know</u>

- How to build an SLR Parser
  - Item Closure Set
  - Item Set GoTo
- Creating an SLR Parser Table
  - Action Table
  - Goto Table
  - Accept / Reject

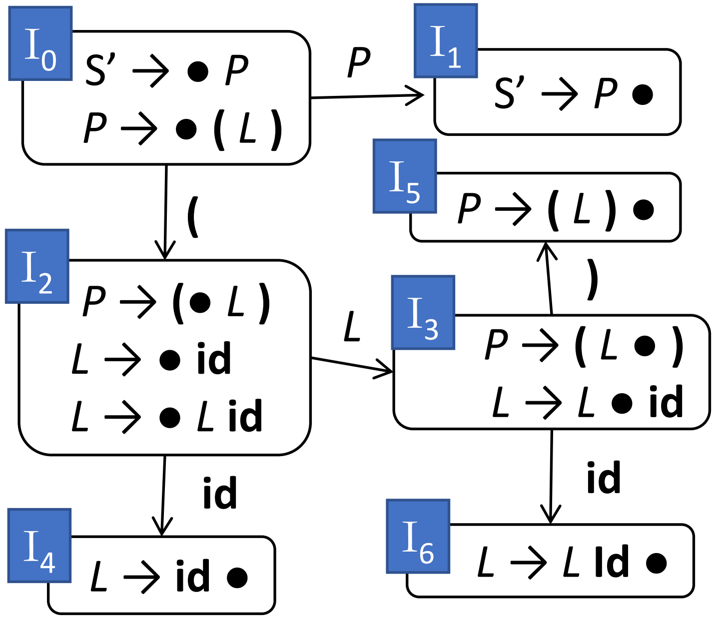
Parsing

Building FSM

LR Parser Construction

Grammar G

1. *S'* ::= *P*
2. *P* ::= (*L*)
3. *L* ::= **id**
4. *L* ::= *L* **id**

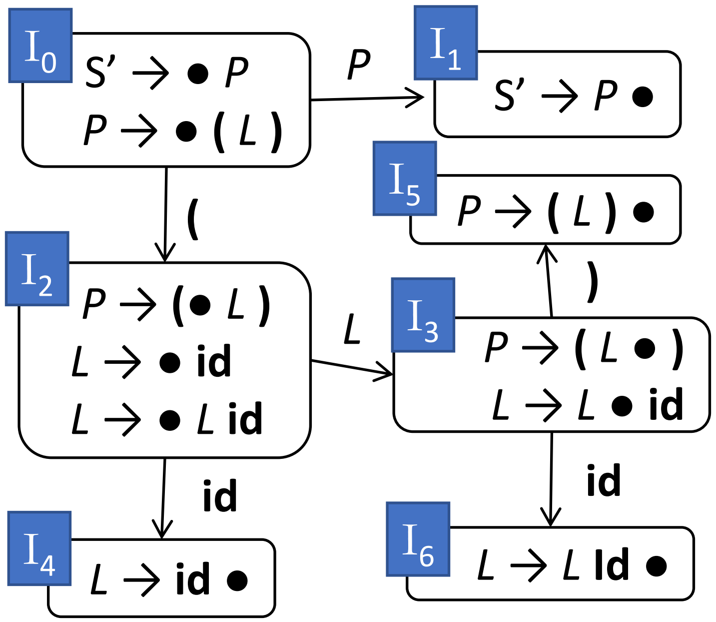


Convert FSM to Table

LR Parser Construction

Grammar G

❶ *S'* ::= *P*
❷ *P* ::= (*L*)
❸ *L* ::= **id**
❹ *L* ::= *L* **id**



	Action Table				GoTo Table	
	(	**)**	**id**	**eof**	***P***	***L***
I_0	S I_2				I_1	
I_1				☺		
I_2			S I_4			I_3
I_3		S I_5	S I_6			
I_4		R❸	R❸			
I_5				R❷		
I_6		R❹	R❹			

Outline

Today's Lecture - Scope

Finish up Parsers

- Running the SLR Parser
- LL(1) and SLR Language limits

Semantics

- Program meaning

Scope

- Name analysis

Parsing

Running the SLR Parser

LR Parser Construction

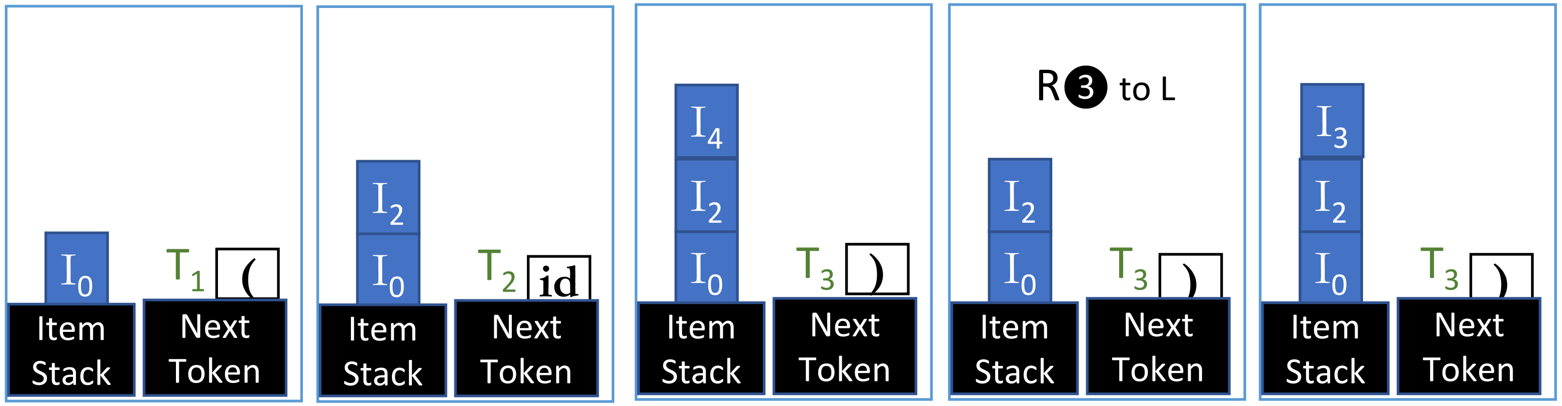
Grammar G

1. $S' ::= P$
2. $P ::= (L)$
3. $L ::=$ **id**
4. $L ::= L$ **id**

Input String

(id) eof

	Action Table: (	)	id	eof	GoTo Table: *P*	*L*
I_0	S I_2				I_1	
I_1				☺		
I_2			S I_4			I_3
I_3		S I_5	S I_6			
I_4		R 3	R 3			
I_5				R 2		
I_6		R 4	R 4			



Running the SLR Parser

LR Parser Construction

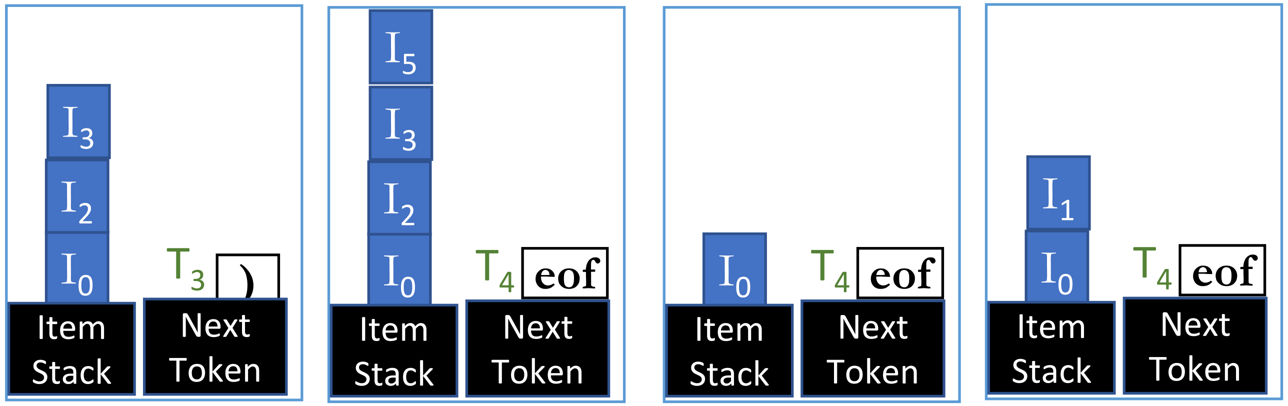
Grammar G

1. $S' ::= P$
2. $P ::= (L)$
3. $L ::= \textbf{id}$
4. $L ::= L\ \textbf{id}$

Input String

(id) eof

	Action Table				GoTo Table	
	(	**)**	**id**	**eof**	***P***	***L***
I_0	S I_2				I_1	
I_1				☺		
I_2			S I_4			I_3
I_3		S I_5	S I_6			
I_4		R 3	R 3			
I_5				R 2		
I_6		R 4	R 4			



Outline

Today's Lecture - Scope

Finish up Parsers

- Running the SLR Parser
- LL(1) and SLR Language limits

Semantics

- Program meaning

Scope

- Name analysis

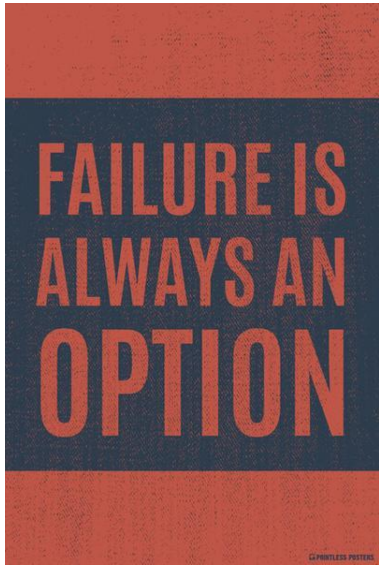
Parsing

When Does the Parser Fail?

LL(1) and SLR Language Limits

For both the LL and LR parsers, two types of failure:

- *Running the parser fails:* The input isn't in the language
- *Building the parser fails:* The language is too expressive



When Running The Parser Fails

LL(1) and SLR Language Limits

The input string is rejected

- Happens whenever either parser table indexes an empty cell
- Happens whenever either parser gets to the end of input without the accept condition

This is the parser working as intended

- Just means the user is at fault with bad input

When Does the Parser Fail?

LL(1) and SLR Language Limits

How building the parser fails

- Happens whenever two entries are in a cell
- For LR parsers, multiple types of collision:
  - Shift/Reduce: a reduce and a shift action in the same cell
  - Reduce/Reduce: reduce by two different productions

This is a problem!

- Means the language isn’t captured by the formalize (e.g. it’s not LL(1), not SLR, whatever)

Bottom-Up SDT

LL(1) and SLR Language Limits

Fairly intuitive

- Add a translation type to each item
- Like LL(1) parser, items are popped right-to left

Terminals translations

- Read lexeme value during a shift

Nonterminal translations

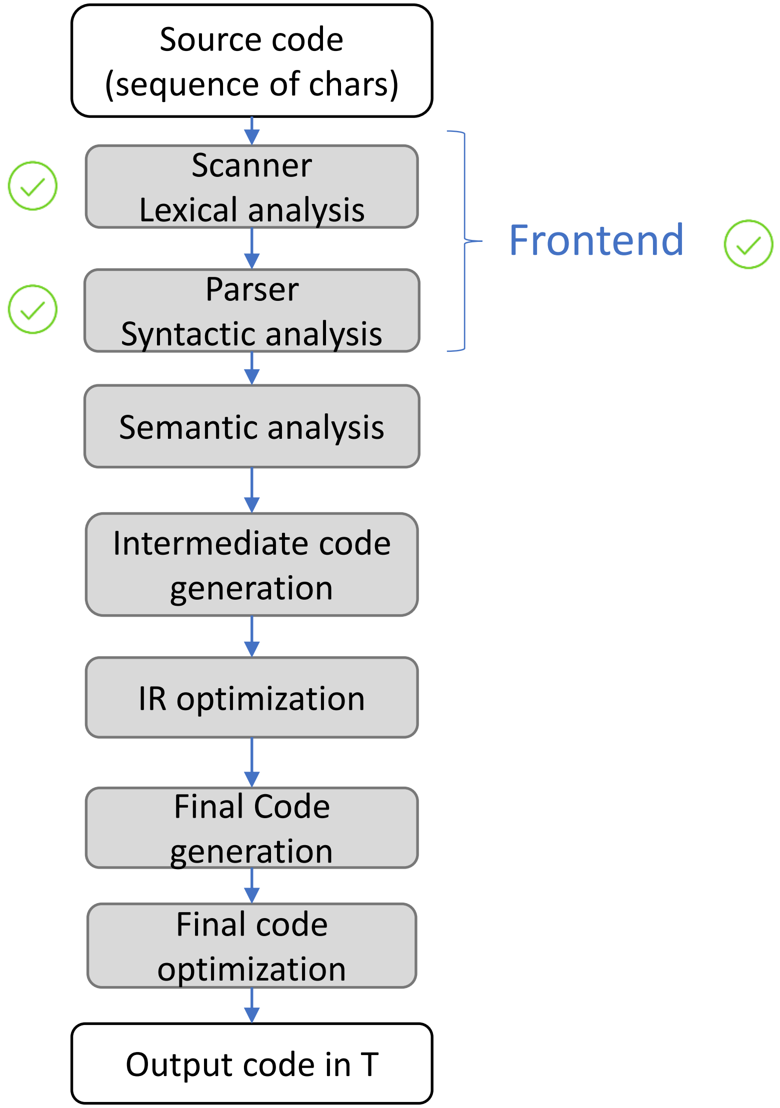
- Read translations of popped RHS symbols

That's all for parsers!

Frontend Finished

ABET Course Outcomes

1. Understanding the role and structure of compilers, and its various phases
2. Constructing an unambiguous grammar for a programming language
3. Generating a lexer and parser using automatic tools
4. Constructing machines to recognize regular expressions (NFA, DFA) and grammars (LL and LR parsers)
5. Generating intermediate form from source code
6. Type checking and static analysis
7. Assembly/binary code generation




COMPILER
LAND
Code Generation
Execution
Runtime Environment
Optimization
Intermediate Representation
SDD
Parsing
Semantics
Lexical Analysis
Syntactic Definiton

Outline

Today's Lecture - Scope

Finish up Parsers

- Running the SLR Parser
- LL(1) and SLR Language limits

Semantics

- Program meaning

Scope

- Name analysis

Parsing

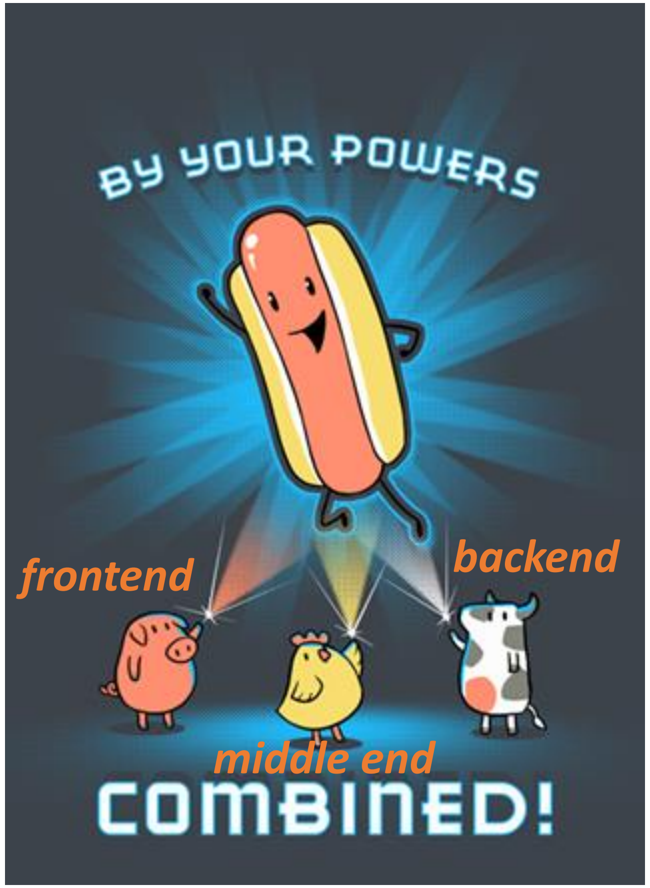
Semantics

Compilers: A Delicious Medley of CS

Today's Lecture - Scope

Learning compilers is kinda like a tasting menu of other CS domains

- Front end – Automata theory / discrete structures
- Middle end – Software Engineering / PL
- Back end – Architecture / Assembly code

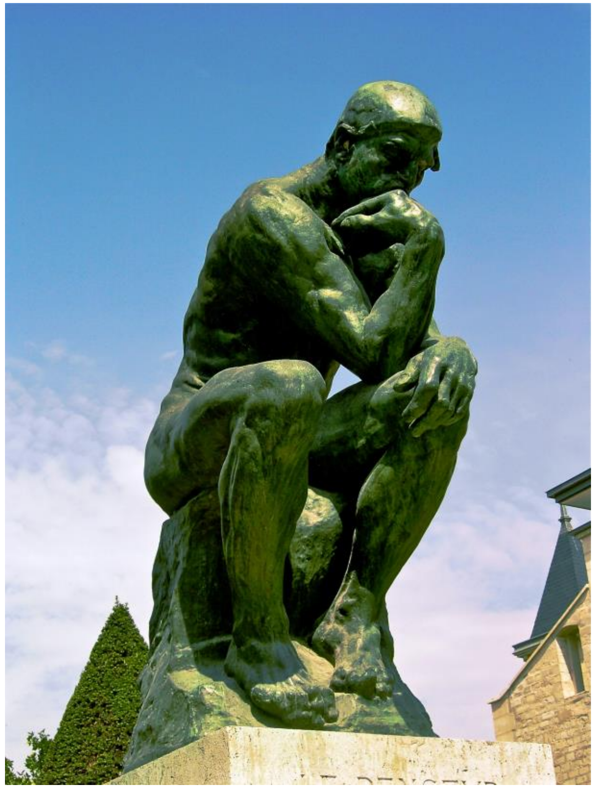


Language Design

Today's Lecture - Scope

Things are about to get a lot more code-y

- Maybe also a bit more cerebral
- Making a compiler empowers you to make a language!
  - How *should* a language be built?



Syntax vs Semantics

Semantic Analysis

Program Syntax

- Does the program have a valid *structure*?

Program Semantics

- Does the program have a valid *meaning*?

Goals

Semantic Analysis

Error Checking

- Is the program's meaning sensible?

Program "Understanding"

- To what does an identifier refer?
- To what operator does a program refer?

Example Program Snippet

```
a + b
```

Is this addition?
String concatenation?
User-defined operation?

Respecting Program Semantics

Semantic Analysis

Compiler must facilitate language semantics

- Prerequisite: Infer the intended program behavior w.r.t. semantics
- Approach: Take multiple passes over the completed AST

One example: scope

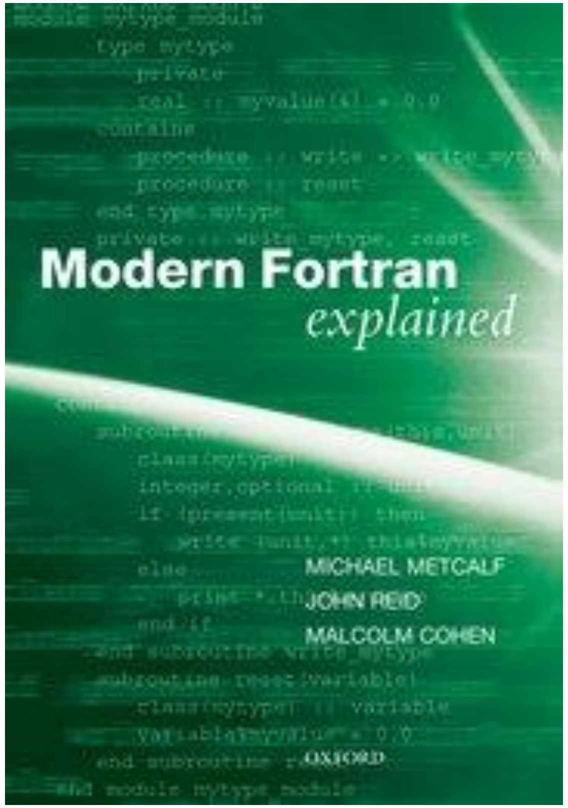
Scope
Semantic Analysis

- A central issue in name analy
  is to determine the **lifetime** c
  variable, function, *etc.*
- Scope definition: the block of
  code in which a name is
  visible/valid

Scope: A Language Feature
Semantic Analysis

- Some languages have NO notion of scope
  - Assembly / FORTRAN
- Most familiar: static / most deeply nested
  - C / C++ / Java

There are several decisions to make, we'll overview a couple of them



Kinds of Scope

Scope Decisions

- Static Scope
  - Most deeply nested w.r.t. **syntactic** block (determined at compile time)
- Dynamic Scope
  - Most deeply nested w.r.t. **calling context** (determined at runtime)

Forward Reference

Scope Decisions

- Do we allow use before name is (lexically) defined?

```
void country() {
    western();
}
void western() {
    country();
}
```

- Requires 2 passes over the program
  - 1 to fill symbol table
  - 1 pass to use symbols

Variable Shadowing

Scope Decisions

- Do we allow names to be re-used?

- What about when the kinds are different?

```
void smoothJazz(int a){
 int a;
 if (a){
   int a;
   if (a){
     int a;
   }
 }
}

void hardRock(int a){
 int hardRock;
}
```

Scope Kind & Shadowing

Scope Decisions

```
int a = 1;
int hop(){
     return a;
}
int hip(){
     int a = 2;
     return hop();
}
int hippo(){
     return hip();
}
```

Overloading
Scope Decisions

- Do we allow same names, same scope, different types?

```
int techno(int a){ … }
bool techno(int a){ … }
bool techno(bool a){ … }
bool techno(bool a, bool b){ }
```

Our Scope Decisions

Scope Decisions

- What scoping rules will we employ?
- What info does the compiler need?