CS 241, Lecture 15 - Type Checking Thurs, Mar 07, 2019

1 Warm Up Problem

Consider the following grammar:

$$S' \to \vdash S \dashv \quad (1)$$
$$S \to aS \quad (2)$$
$$S \to B \quad (3)$$
$$B \to aBb \quad (4)$$
$$B \to \epsilon \quad (5)$$

Draw the SLR(1) transducer (and hence also an LR(0) transducer) for this grammar. Is it LR(0)? What about SLR(1)?

We see in the diagram that it is NOT LR(0), there are shift-reduce conflicts. We remedy those by checking the Follow in states 2 and 8.

2 Creating a Parse Tree

Building the Parse Tree:

- The difference between top-down parsing and bottom-up parsing is that in top-down, you pop a symbol S from the stack and push the next rule, and make the new symbols the children of the popped symbol. With bottom-up parsing, you reduce symbols into another rule, and keep the old symbols as the children.
- For example, for bottom-up, if we had the grammar:

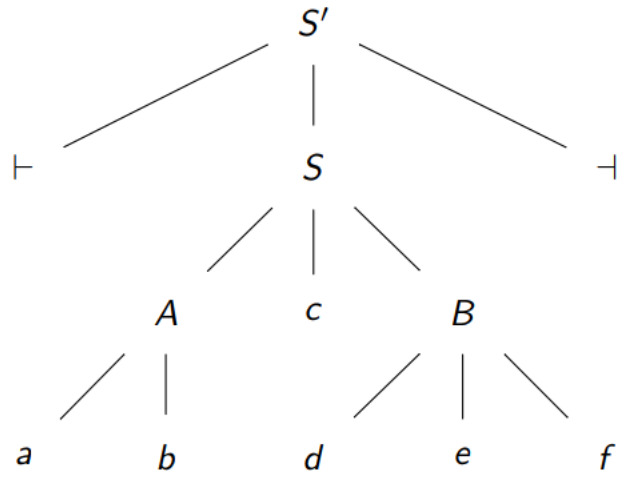
$$S' \to \vdash S \dashv \quad (0)$$
$$S \to AcB \quad (1)$$
$$A \to ab \quad (2)$$
$$A \to ff \quad (3)$$
$$B \to def \quad (4)$$
$$B \to ef \quad (5)$$

Then for a string $\vdash abcdef \dashv$:

Stack	Read	Processing	Action
	ϵ	$\vdash abcdef \dashv$	Shift $\vdash$
$\vdash$	$\vdash$	$abcdef \dashv$	Shift a
$\vdash a$	$\vdash a$	$bcdef \dashv$	Shift b
$\vdash ab$	$\vdash ab$	$cdef \dashv$	Reduce (2); pop b, a, push A
$\vdash A$	$\vdash ab$	$cdef \dashv$	Shift c
$\vdash Ac$	$\vdash abc$	$def \dashv$	Shift d
$\vdash Acd$	$\vdash abcd$	$ef \dashv$	Shift e
$\vdash Acde$	$\vdash abcde$	$f \dashv$	Shift f
$\vdash Acdef$	$\vdash abcdef$	$\dashv$	Reduce (4); pop f, d, e push B
$\vdash AcB$	$\vdash abcdef$	$\dashv$	Reduce (1); pop B, c, A push S
$\vdash S$	$\vdash abcdef$	$\dashv$	Shift $\dashv$
$\vdash S \dashv$	$\vdash abcdef \dashv$	ϵ	Reduce (0); pop $\dashv, S, \vdash$ push S'
S'	$\vdash abcdef \dashv$	ϵ	Accept

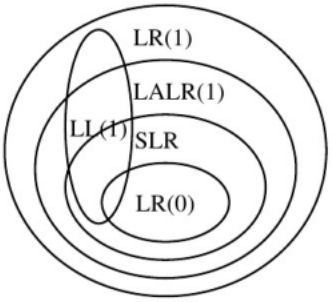
We get the following parse tree:

- $\vdash$ (shift)
- $\vdash a$ (shift)
- $\vdash ab$ (shift)
- $\vdash {}_aA_b$ (Reduce)
- $\vdash {}_aA_bc$ (Shift)
- $\vdash {}_aA_bcd$ (Shift)
- $\vdash {}_aA_bcde$ (Shift)
- $\vdash {}_aA_bcdef$ (Shift)
- $\vdash {}_aA_bc\,{}^{B}_{def}$ (Reduce)
- $\vdash {}^{S}_{{}_aA_bc\,{}^{B}_{def}}$ (Reduce)
- $\vdash {}^{S}_{{}_aA_bc\,{}^{B}_{def}} \dashv$ (Shift)
- Reduce on right



- We get this image for how grammars are classified:

LL, SLR, LR, LALR Grammars



3 Parser and Assignment Information

- Our parser will output a .wlp4i file. For example:

$$S \to BOFeEOF$$
$$e \to e + t$$
$$e \to t$$
$$t \to ID$$

with input $BOFa + b + cEOF$, where a, b, c are IDs, then you will get the file:

```
S BOF e EOF
BOF BOF
e e + t
e e + t
e t
t ID
ID a
+ +
t ID
ID b
+ +
t ID
ID c
EOF EOF
```

- As an overview:
  - A6 is WLP4 text file to WLP4 tokens and lexemes
  - A7 is WLP4 tokens and lexemes to a parse tree

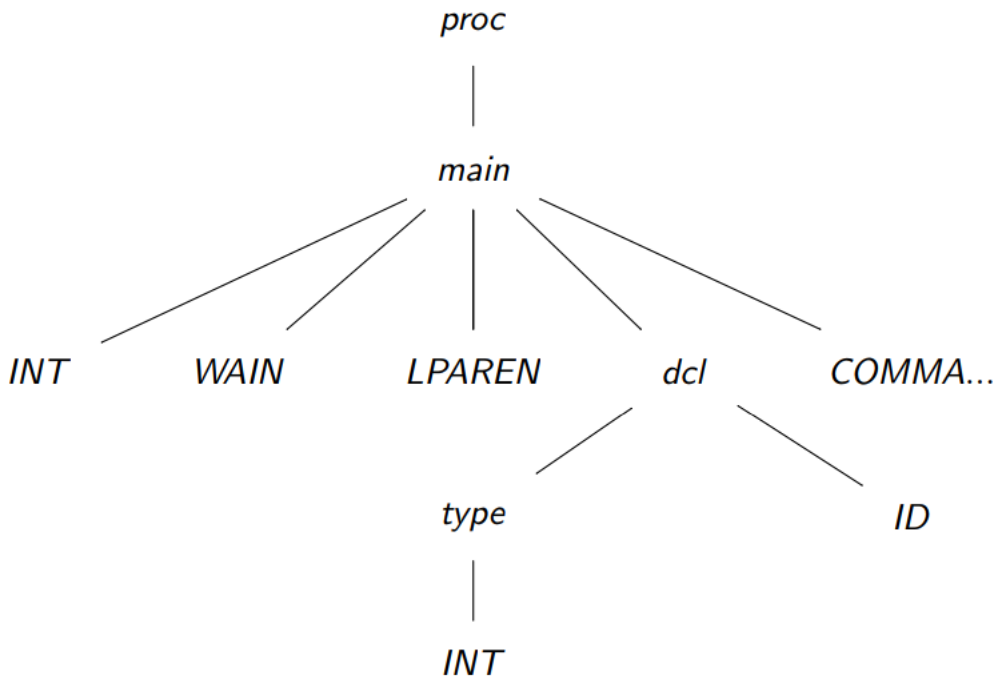
- A8 is a parse tree to augmented parse tree + symbol tabels
- A9 and A10 is augmented parse trees to MIPS assembly

4 Context Sensitive Analysis / Type Checking

- Evidently, we cannot enforce everything with a CFG. For example:
  - Type checking
  - Declaration before use
  - Scoping
  - Well-typed expressions

  This is where we move to context-sensitive grammars.

- Let's use the following parse tree for our analysis for now...



- In code, you could use a *very* simple tree object:

```
class Tree {
    public:
        string rule;
        vector <string> tokens;
        vector <Tree> children;
};
```

- How could we determine multiple/missing declaration errors? Use a symbol table!
- We don't have to pass through the file twice - this is because we cannot use variables before they are declared!
- We need to keep track of *functions* and *variables*!
- Use a map that tracks functions (global) and a map for each function declaration if that is valid!
- You may need to keep a global variable that tracks which procedure we're in.
- For functions, we *only* need to store the parameters, as WLP4 is limited to int type only for functions.
- We do need to be careful - we must ensure that the parameters passed in are the *correct* parameters - we don't want int pointers if we want ints!
- Symbol tables should be a procedure name, pairs of signatures, and symbol tables. This could be a map<string, pair<vector<string>, map<string, string >>>
- Likewise, unlike rustcc, since this has support for pointers, we must keep track of those as well. We have two types: ints and pointers to ints. For type checking, upon delcaration, we add it to the symbol table.
- So for example, given the following code:

```
int f() {
    int *a = NULL;
    return 9;
}

int wain(int a, int b) {
    int x = 10;
    return x + a + b;
}
```

  this would return a symbol table with the two entries:

  - f <>, <a ->, int*>
  - wain<int, int>, and <a -> int, b -> int, x -> int>

- So, how do we actually catch a type error?
  - First, we figure out the type of every epxression using type rules.
  - Then, if none such rule exists or the types do not match a given rule, produce an error.

- We use inference rules from 245 - if an ID is declared with type τ then it has type int if there is a NUM, and int* if it there is a NULL. That is, $\frac{<id.name,\tau> \in declarations}{id.name:\tau}$.
- $\frac{E:\tau}{(E):\tau}$
- $\frac{E:int}{\&E:int*}$
- $\frac{E:int*}{*E:int}$
- $\frac{E:int}{newint[E]:int*}$
- $\frac{E_1:int, E_2:int}{E_1 + - / * \% E_2:int}$
- $\frac{E_1:int*, E_2:int}{E_1 + - E_2:int*}$
- $\frac{E_1:int, E_2:int*}{E_1 + - E_2:int*}$
- $\frac{E_1:int*, E_2:int*}{E_1 - E_2:int*}$