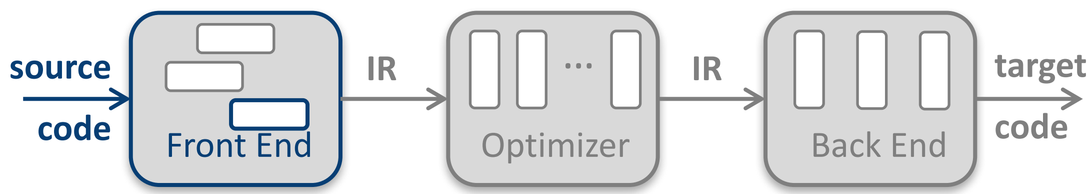
COMP 412
FALL 20167

Computing Inside The Parser

— Syntax-Directed Translation, II —

Comp 412



Chapter 4 in EaC2e

Midterm Exam

When? Where?

- Wednesday, October 18, 2017 in Keck 100
- Two hour, closed book, closed notes, closed devices exam
- I had said 7PM. Will 7:30PM work?

You are responsible for material from:

- Lectures from start of classes through today
- Chapters 1 through 5 in EaC2e
  - Excluding attribute grammar material (§ 4.3)

Example

Computing the value of an unsigned integer

Consider the simple grammar

1	*Number*	→	digit *DigitList*
2	*DigitList*	→	digit *DigitList*
3		|	epsilon

One obvious use of the grammar is to convert an **ASCII** string of the number to its integer value

- Build computation into parser
- An easy intro to *syntax-directed translation*

```
pair (boolean, int)  Number( value ) {
   if (word = digit) then {
      value = ValueOf( digit );
      word = NextWord( );
      return DigitList( value );
   }
   else return (false, invalid value);
}

pair (boolean, int)  DigitList( value ) {
   if (word = digit) then {
      value = value * 10 + ValueOf( digit );
      word = NextWord( );
      return DigitList( value );
   }
   else return (true, value);
}
```

SDT in a top-down, recursive-descent parser

SDT in a Bottom-Up Parser (*e.g.*, **LR(1)**)

We specify SDT actions relative to the syntax

1	*Number*	→	*DigitList*	{ value(**LHS**) = value(*DL*) ; }
2	*DigitList*	→	*DigitList* digit	{ value(**LHS**) = value(*DL*) * 10 + value(*digit*) }
3		|	digit	{ value(**LHS**) = value(*digit*); }

The compiler writer can attach "actions" to productions and have those actions execute each time the parser reduces by that production

- Simple mechanism that ties computation to syntax
- Needs storage associated with each symbol in each production
- Needs a scheme to name that storage

SDT in Bison or Yacc

In Bison or Yacc, we specify SDT actions using a simple notation

1	*Number*	→	*DigitList*	{ \$\$ = \$1; }
2	*DigitList*	→	*DigitList* <u>digit</u>	{ \$\$ = \$1 * 10 + \$2; }
3		|	<u>digit</u>	{ \$\$ = \$1; }

Scanner provides the value of *digit.*

The compiler writer provides production-specific code snippets that execute when the parser reduces by that production

- Positional notation for the value associated with a symbol
  - \$\$ is the **LHS**; \$1 is the first symbol in the **RHS**; \$2 the second, …
- Compiler writer can put arbitrary code in these snippets
  - Solve a travelling salesman problem, compute **PI** to 100 digits, …
  - More importantly, they can compute on the lexemes of grammar symbols and on information derived and stored earlier in translation

How does this fit into the LR(1) skeleton parser?

Fitting AHSDT into the LR(1) Skeleton Parser

```
stack.push(INVALID);
stack.push(s0);                       // initial state
word = scanner.next_word();
loop forever {
        s = stack.top();
        if ( ACTION[s,word] == "reduce A→β" ) then {
          stack.popnum(2*|β|);   // pop 2*|β| symbols
          s = stack.top();
          stack.push(A);                // push LHS, A
          stack.push(GOTO[s,A]);  // push next state
        }
        else if ( ACTION[s,word] == "shift si" ) then {
                stack.push(word); stack.push(si);
                word ← scanner.next_word();
        }
        else if ( ACTION[s,word] == "accept"
                                & word == EOF )
                then break;
        else throw a syntax error;
}
report success;
```

Actions are taken on reductions

- Insert a call to *Work*() before the call to stack.popnum()
- *Work*() contains a case statement that switches on the production number
- Code in *Work*() can read items from the stack
  - → That is why it calls *Work*() before stack.popnum()

Fitting AHSDT into the LR(1) Skeleton Parser

```
stack.push(INVALID);
stack.push(s0);                    // initial state
word = scanner.next_word();
loop forever {
    s = stack.top();
    if ( ACTION[s,word] == "reduce A→β" ) then {
      stack.popnum(2*|β|);   // pop 2*|β| symbols
      s = stack.top();
      stack.push(A);                // push LHS, A
      stack.push(GOTO[s,A]);  // push next state
    }
    else if ( ACTION[s,word] == "shift si" ) then {
        stack.push(word); stack.push(si);
        word ← scanner.next_word();
    }
    else if ( ACTION[s,word] == "accept"
                        & word == EOF )
        then break;
    else throw a syntax error;
}
report success;
```

Passing values between actions

- Tie values to instances of grammar symbols
  - → Equivalent to parse tree nodes
- We can pass values on the stack
  - → Push / pop 3 rather than 2
  - → *Work*() takes the stack as input (*conceptually*) and returns the value for the reduction it processes
  - → *Shift* creates initial values

Fitting AHSDT into the LR(1) Skeleton Parser

```
stack.push(INVALID);
stack.push(s0);                       // initial state
word = scanner.next_word();
loop forever {
        s = stack.top();
        if ( ACTION[s,word] == "reduce A→β" ) then {
          r = Work(stack, "A→β")
          stack.popnum(3*|β|);  // pop 3*|β| symbols
          s = stack.top();            // save exposed state
          stack.push(A);              // push A
          stack.push (r);             // push result of WORK()
          stack.push(GOTO[s,A]);  // push next state
        }
        else if ( ACTION[s,word] == "shift si" ) then {
                stack.push(word);
                stack.push(Initial Value);
                stack.push(si);
                word ← scanner.next_word();
        }
        else if ( ACTION[s,word] == "accept"
                              & word == EOF )
                then break;
        else throw a syntax error;
}
report success;
```

- Modifications are minor
  - Insert call to *Work*()
  - Change the push() & pop() behavior
- Same asymptotic behavior as the original algorithm.
  - 50% more stack space
- Last obstacle is making it easy to write the code for *Work()*

*Note that, in **C**, the stack has some odd union type.*

Translating Code Snippets Into *Work*()

For each production, the compiler writer can provide a code snippet

{ *value* = *value* * *10* + *digit*; }

We need a scheme to name stack locations. Yacc introduced a simple one that has been widely adopted.

- $$ refers to the result, which will be pushed on the stack
- $1 is the first item on the productions right hand side
- $2 is the second item
- $3 is the third item, and so on …

The digits example above becomes

{ $$ = $1 * 10 + $2; }

Translating Code Snippets Into *Work()*

How do we implement *Work()* ?

- *Work()* takes 2 arguments: the stack and a production number
- *Work()* contains a case statement that switches on production number
  - Each case contains the code snippet for a reduction by that production
  - The $1, $2, $3 … macros translate into references into the stack
  - The $$ macro translates into the return value

```
…
if ( ACTION[s,word] == "reduce A→β" ) then {
        r = Work(stack, "A→β")
        stack.popnum(3*|β|);  // pop 3*|β| symbols
        s = stack.top();       // save exposed state
        stack.push(A);         // push A
        stack.push (r);        // push result of WORK()
        stack.push(GOTO[s,A]); // push next state
    }
…
```

$$ translates to **r**

$*i* translates to the stack location 3 *(|*β*| - *i* + 1) units down from stacktop

Note that *β*, *i*, *3*, and *1* are all constants so $*i* can be evaluated to a compile-time constant

SDT is a Mechanism

AHSDT allows the compiler writer to specify syntax-driven computations

- Build an **IR** representation
  - Critical for later phases of the compiler
  - Might execute it directly to create an interpreter
    - → *Calculators, accounting systems, command-line shells, …*
  - Might analyze it to derive knowledge
    - → *Optimizing compilers, theorem provers, …*
  - Might transform it and re-generate source code
    - → *Automatic parallelization systems, code refactoring tools, …*
- Measure and report on program properties
- Correct syntax errors

Key point: the mechanism does not determine how you use it

Corollary: the syntax does not dictate the form or content of the **IR**

Example — Building an Abstract Syntax Tree

1	*Goal*	$\rightarrow$	*Expr*	$$ = $1;
2	*Expr*	$\rightarrow$	*Expr* + *Term*	$$ = MakeAddNode($1,$3);
3		|	*Expr* - *Term*	$$ = MakeSubNode($1,$3);
4		|	*Term*	$$ = $1;
5	*Term*	$\rightarrow$	*Term* * *Factor*	$$ = MakeMulNode($1,$3);
6		|	*Term* / *Factor*	$$ = MakeDivNode($1,$3);
7		|	*Factor*	$$ = $1;
8	*Factor*	$\rightarrow$	(*Expr*)	$$ = $2;
9		|	number	$$ = MakeNumNode(token);
10		|	ident	$$ = MakeIdNode(token);

Assumptions:

- constructors for each node
- stack holds pointers to nodes

We call $$ = $1; a copy rule.

Example — Building an Abstract Syntax Tree

Stack	Input	Action
$	id – num * id	*shift*
$ id	– num * id	*reduce 10*
$ *Factor*	– num * id	*reduce 7*
$ *Term*	– num * id	*reduce 4*
$ *Expr*	– num * id	*shift*
$ *Expr* –	num * id	*shift*
$ *Expr* – num	* id	*reduce 9*
$ *Expr* – *Factor*	* id	*reduce 7*
$ *Expr* – *Term*	* id	*shift*
$ *Expr* – *Term* *	id	*shift*
$ *Expr* – *Term* * id		*reduce 10*
$ *Expr* – *Term* * *Factor*		*reduce 5*
$ *Expr* – *Term*		*reduce 3*
$ *Expr*		*accept*

Consider "x – 2 * y"

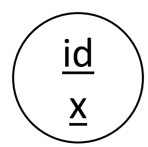
- Trace of the LR(1) parse
- Detail of states abstracted

Expression Grammar

1	*Goal*	→	*Expr*
2	*Expr*	→	*Expr* + *Term*
3		|	*Expr* - *Term*
4		|	*Term*
5	*Term*	→	*Term* * *Factor*
6		|	*Term* / *Factor*
7		|	*Factor*
8	*Factor*	→	(*Expr*)
9		|	num
10		|	id

Example — Building an Abstract Syntax Tree

Stack	Input	Action
$	id – num * id	*shift*
$ id	– num * id	*reduce 10*
$ *Factor*	– num * id	*reduce 7*
$ *Term*	– num * id	*reduce 4*
$ *Expr*	– num * id	*shift*
$ *Expr* –	num * id	*shift*
$ *Expr* – num	* id	*reduce 9*
$ *Expr* – *Factor*	* id	*reduce 7*
$ *Expr* – *Term*	* id	*shift*
$ *Expr* – *Term* *	id	*shift*
$ *Expr* – *Term* * id		*reduce 10*
$ *Expr* – *Term* * *Factor*		*reduce 5*
$ *Expr* – *Term*		*reduce 3*
$ *Expr*		*accept*



Example — Building an Abstract Syntax Tree

Stack	Input	Action
$	id – num * id	*shift*
$ id	– num * id	*reduce 10*
$ *Factor*	– num * id	*reduce 7*
$ *Term*	– num * id	*reduce 4*
$ *Expr*	– num * id	*shift*
$ *Expr* –	num * id	*shift*
$ *Expr* – num	* id	*reduce 9*
$ *Expr* – *Factor*	* id	*reduce 7*
$ *Expr* – *Term*	* id	*shift*
$ *Expr* – *Term* *	id	*shift*
$ *Expr* – *Term* * id		*reduce 10*
$ *Expr* – *Term* * *Factor*		*reduce 5*
$ *Expr* – *Term*		*reduce 3*
$ *Expr*		*accept*

Action is a copy rule

id
x

Example — Building an Abstract Syntax Tree

Stack	Input	Action
$	id – num * id	*shift*
$ id	– num * id	*reduce 10*
$ *Factor*	– num * id	*reduce 7*
$ *Term*	– num * id	*reduce 4*
$ *Expr*	– num * id	*shift*
$ *Expr* –	num * id	*shift*
$ *Expr* – num	* id	*reduce 9*
$ *Expr* – *Factor*	* id	*reduce 7*
$ *Expr* – *Term*	* id	*shift*
$ *Expr* – *Term* *	id	*shift*
$ *Expr* – *Term* * id		*reduce 10*
$ *Expr* – *Term* * *Factor*		*reduce 5*
$ *Expr* – *Term*		*reduce 3*
$ *Expr*		*accept*

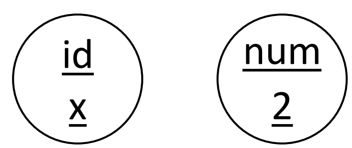
id
x

num
2

Example — Building an Abstract Syntax Tree

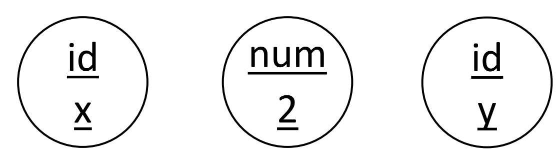
Stack	Input	Action
$	id – num * id	*shift*
$ id	– num * id	*reduce 10*
$ *Factor*	– num * id	*reduce 7*
$ *Term*	– num * id	*reduce 4*
$ *Expr*	– num * id	*shift*
$ *Expr* –	num * id	*shift*
$ *Expr* – num	* id	*reduce 9*
$ *Expr* – *Factor*	* id	*reduce 7*
$ *Expr* – *Term*	* id	*shift*
$ *Expr* – *Term* *	id	*shift*
$ *Expr* – *Term* * id		*reduce 10*
$ *Expr* – *Term* * *Factor*		*reduce 5*
$ *Expr* – *Term*		*reduce 3*
$ *Expr*		*accept*

Action is a copy rule



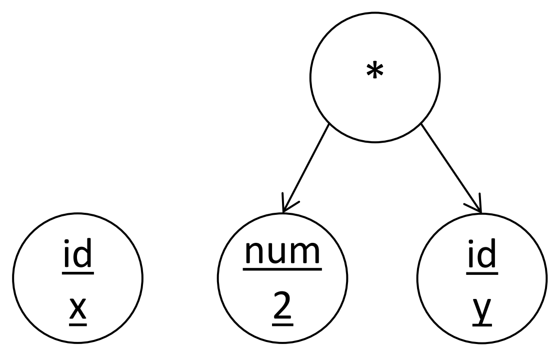
Example — Building an Abstract Syntax Tree

Stack	Input	Action
$	id – num * id	*shift*
$ id	– num * id	*reduce 10*
$ *Factor*	– num * id	*reduce 7*
$ *Term*	– num * id	*reduce 4*
$ *Expr*	– num * id	*shift*
$ *Expr* –	num * id	*shift*
$ *Expr* – num	* id	*reduce 9*
$ *Expr* – *Factor*	* id	*reduce 7*
$ *Expr* – *Term*	* id	*shift*
$ *Expr* – *Term* *	id	*shift*
$ *Expr* – *Term* * id		*reduce 10*
$ *Expr* – *Term* * *Factor*		*reduce 5*
$ *Expr* – *Term*		*reduce 3*
$ *Expr*		*accept*



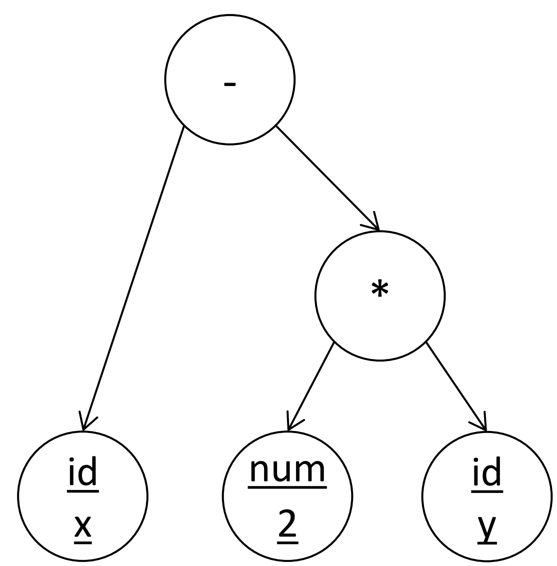
Example — Building an Abstract Syntax Tree

Stack	Input	Action
$	id – num * id	*shift*
$ id	– num * id	*reduce 10*
$ *Factor*	– num * id	*reduce 7*
$ *Term*	– num * id	*reduce 4*
$ *Expr*	– num * id	*shift*
$ *Expr* –	num * id	*shift*
$ *Expr* – num	* id	*reduce 9*
$ *Expr* – *Factor*	* id	*reduce 7*
$ *Expr* – *Term*	* id	*shift*
$ *Expr* – *Term* *	id	*shift*
$ *Expr* – *Term* * id		*reduce 10*
$ *Expr* – *Term* * *Factor*		*reduce 5*
$ *Expr* – *Term*		*reduce 3*
$ *Expr*		*accept*



Example — Building an Abstract Syntax Tree

Stack	Input	Action
$	id – num * id	*shift*
$ id	– num * id	*reduce 10*
$ *Factor*	– num * id	*reduce 7*
$ *Term*	– num * id	*reduce 4*
$ *Expr*	– num * id	*shift*
$ *Expr* –	num * id	*shift*
$ *Expr* – num	* id	*reduce 9*
$ *Expr* – *Factor*	* id	*reduce 7*
$ *Expr* – *Term*	* id	*shift*
$ *Expr* – *Term* *	id	*shift*
$ *Expr* – *Term* * id		*reduce 10*
$ *Expr* – *Term* * *Factor*		*reduce 5*
$ *Expr* – *Term*		*reduce 3*
$ *Expr*		*accept*

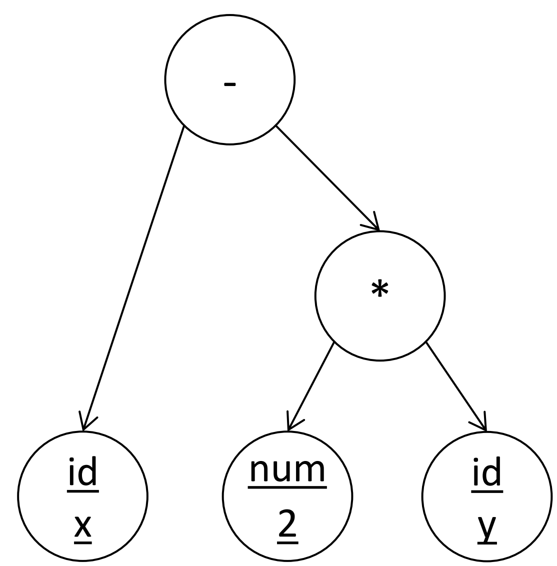


Example — Building an Abstract Syntax Tree

Stack	Input	Action
$	id – num * id	*shift*
$ id	– num * id	*reduce 10*
$ *Factor*	– num * id	*reduce 7*
$ *Term*	– num * id	*reduce 4*
$ *Expr*	– num * id	*shift*
$ *Expr* –	num * id	*shift*
$ *Expr* – num	* id	*reduce 9*
$ *Expr* – *Factor*	* id	*reduce 7*
$ *Expr* – *Term*	* id	*shift*
$ *Expr* – *Term* *	id	*shift*
$ *Expr* – *Term* * id		*reduce 10*
$ *Expr* – *Term* * *Factor*		*reduce 5*
$ *Expr* – *Term*		*reduce 3*
$ *Expr*		*accept*

AHSDT Works!

- Built the **AST**
- Some reduce actions just copied values; others called constructors.
- Same tree as earlier slide



Example — Emitting **ILOC**

1	*Goal*	$\rightarrow$	*Expr*	
2	*Expr*	$\rightarrow$	*Expr* +*Term*	$$ = *NextRegister*(); *Emit*(add, $1, $3, $$);
3		|	*Expr* - *Term*	$$ = *NextRegister*(); *Emit*(sub, $1, $3, $$);
4		|	*Term*	$$ = $1;
5	*Term*	$\rightarrow$	*Term* * *Factor*	$$ = *NextRegister*(); *Emit*(mult, $1, $3, $$)
6		|	*Term* / *Factor*	$$ = *NextRegister*(); *Emit*(div, $1, $3, $$);
7		|	*Factor*	$$ = $1;
8	*Factor*	$\rightarrow$	<u>(</u> *Expr* <u>)</u>	$$ = $2;
9		|	<u>number</u>	$$ = *NextRegister*(); *Emit*(loadI,Value(lexeme),$$);
10		|	<u>ident</u>	$$ = *NextRegister*(); *EmitLoad*(ident,$$);

Assumptions

- *NextRegister()* returns a virtual register name
- *Emit()* can format assembly code
- *EmitLoad()* handles addressability & gets a value into a register

Copy rules on the same productions as in the last example

Same parse, different rule set

Example — Emitting ILOC

Stack	Input	Action
$	id – num * id	*shift*
$ id	– num * id	*reduce 10*
$ *Factor*	– num * id	*reduce 7*
$ *Term*	– num * id	*reduce 4*
$ *Expr*	– num * id	*shift*
$ *Expr* –	num * id	*shift*
$ *Expr* – num	* id	*reduce 9*
$ *Expr* – *Factor*	* id	*reduce 7*
$ *Expr* – *Term*	* id	*shift*
$ *Expr* – *Term* *	id	*shift*
$ *Expr* – *Term* * id		*reduce 10*
$ *Expr* – *Term* * *Factor*		*reduce 5*
$ *Expr* – *Term*		*reduce 3*
$ *Expr*		*accept*

Emitting ILOC

- The actions that generated AST leaves and the actions that generated AST interior nodes emit code.
- pr0 is a base address
- *NextRegister*() returned the next virtual register

loadAI pr0, @x ⇒ vr1

Example — Emitting ILOC

Stack	Input	Action
$	id – num * id	*shift*
$ id	– num * id	*reduce 10*
$ *Factor*	– num * id	*reduce 7*
$ *Term*	– num * id	*reduce 4*
$ *Expr*	– num * id	*shift*
$ *Expr* –	num * id	*shift*
$ *Expr* – num	* id	*reduce 9*
$ *Expr* – *Factor*	* id	*reduce 7*
$ *Expr* – *Term*	* id	*shift*
$ *Expr* – *Term* *	id	*shift*
$ *Expr* – *Term* * id		*reduce 10*
$ *Expr* – *Term* * *Factor*		*reduce 5*
$ *Expr* – *Term*		*reduce 3*
$ *Expr*		*accept*

Action is a copy rule

loadAI pr0, @x ⇒ vr1

Example — Emitting ILOC

Stack	Input	Action
$	id – num * id	*shift*
$ id	– num * id	*reduce 10*
$ *Factor*	– num * id	*reduce 7*
$ *Term*	– num * id	*reduce 4*
$ *Expr*	– num * id	*shift*
$ *Expr* –	num * id	*shift*
$ *Expr* – num	* id	*reduce 9*
$ *Expr* – *Factor*	* id	*reduce 7*
$ *Expr* – *Term*	* id	*shift*
$ *Expr* – *Term* *	id	*shift*
$ *Expr* – *Term* * id		*reduce 10*
$ *Expr* – *Term* * *Factor*		*reduce 5*
$ *Expr* – *Term*		*reduce 3*
$ *Expr*		*accept*

```
loadAI   pr0, @x   ⇒ vr1
loadI    2         ⇒ vr2
```

Example — Emitting ILOC

Stack	Input	Action
$	id – num * id	*shift*
$ id	– num * id	*reduce 10*
$ *Factor*	– num * id	*reduce 7*
$ *Term*	– num * id	*reduce 4*
$ *Expr*	– num * id	*shift*
$ *Expr* –	num * id	*shift*
$ *Expr* – num	* id	*reduce 9*
$ *Expr* – *Factor*	* id	*reduce 7*
$ *Expr* – *Term*	* id	*shift*
$ *Expr* – *Term* *	id	*shift*
$ *Expr* – *Term* * id		*reduce 10*
$ *Expr* – *Term* * *Factor*		*reduce 5*
$ *Expr* – *Term*		*reduce 3*
$ *Expr*		*accept*

Action is a copy rule

```
loadAI   pr0, @x   ⇒ vr1
loadI    2         ⇒ vr2
```

Example — Emitting ILOC

Stack	Input	Action
$	id – num * id	*shift*
$ id	– num * id	*reduce 10*
$ *Factor*	– num * id	*reduce 7*
$ *Term*	– num * id	*reduce 4*
$ *Expr*	– num * id	*shift*
$ *Expr* –	num * id	*shift*
$ *Expr* – num	* id	*reduce 9*
$ *Expr* – *Factor*	* id	*reduce 7*
$ *Expr* – *Term*	* id	*shift*
$ *Expr* – *Term* *	id	*shift*
$ *Expr* – *Term* * id		*reduce 10*
$ *Expr* – *Term* * *Factor*		*reduce 5*
$ *Expr* – *Term*		*reduce 3*
$ *Expr*		*accept*

loadAI	pr0, @x	⇒ vr1
loadI	2	⇒ vr2
loadAI	pr0, @y	⇒ vr3

Example — Emitting ILOC

Stack	Input	Action
$	id – num * id	*shift*
$ id	– num * id	*reduce 10*
$ *Factor*	– num * id	*reduce 7*
$ *Term*	– num * id	*reduce 4*
$ *Expr*	– num * id	*shift*
$ *Expr* –	num * id	*shift*
$ *Expr* – num	* id	*reduce 9*
$ *Expr* – *Factor*	* id	*reduce 7*
$ *Expr* – *Term*	* id	*shift*
$ *Expr* – *Term* *	id	*shift*
$ *Expr* – *Term* * id		*reduce 10*
$ *Expr* – *Term* * *Factor*		*reduce 5*
$ *Expr* – *Term*		*reduce 3*
$ *Expr*		*accept*

loadAI	pr0, @x	⇒ vr1
loadI	2	⇒ vr2
loadAI	pr0, @y	⇒ vr3
mult	vr2, vr3	⇒ vr4

Example — Emitting ILOC

Stack	Input	Action
$	id – num * id	*shift*
$ id	– num * id	*reduce 10*
$ *Factor*	– num * id	*reduce 7*
$ *Term*	– num * id	*reduce 4*
$ *Expr*	– num * id	*shift*
$ *Expr* –	num * id	*shift*
$ *Expr* – num	* id	*reduce 9*
$ *Expr* – *Factor*	* id	*reduce 7*
$ *Expr* – *Term*	* id	*shift*
$ *Expr* – *Term* *	id	*shift*
$ *Expr* – *Term* * id		*reduce 10*
$ *Expr* – *Term* * *Factor*		*reduce 5*
$ *Expr* – *Term*		*reduce 3*
$ *Expr*		*accept*

loadAI	pr0, @x	⇒ vr1
loadI	2	⇒ vr2
loadAI	pr0, @y	⇒ vr3
mult	vr2, vr3	⇒ vr4
sub	vr1,vr4	⇒ vr5

Example — Emitting ILOC

Stack	Input	Action
$	id – num * id	*shift*
$ id	– num * id	*reduce 10*
$ *Factor*	– num * id	*reduce 7*
$ *Term*	– num * id	*reduce 4*
$ *Expr*	– num * id	*shift*
$ *Expr* –	num * id	*shift*
$ *Expr* – num	* id	*reduce 9*
$ *Expr* – *Factor*	* id	*reduce 7*
$ *Expr* – *Term*	* id	*shift*
$ *Expr* – *Term* *	id	*shift*
$ *Expr* – *Term* * id		*reduce 10*
$ *Expr* – *Term* * *Factor*		*reduce 5*
$ *Expr* – *Term*		*reduce 3*
$ *Expr*		*accept*

Emitting ILOC

- Simple, but clean, code
- *EmitLoad()* hides all of the messy details of naming, addressability, and address modes

```
loadAI   pr0, @x   ⇒ vr1
loadI    2         ⇒ vr2
loadAI   pr0, @y   ⇒ vr3
mult     vr2, vr3  ⇒ vr4
sub      vr1,vr4   ⇒ vr5
```

What About *EmitLoad*()

***EmitLoad*() hides lots of details**

- Needs to map an <u>ident</u> to a location
  - Register, symbolic address, or formula to compute a virtual address
  - Implies that the <u>ident</u> has allocated storage
  - Someone or something needs to lay out the contents of data memory
- Needs to generate code to compute the address
  - Register becomes direct reference
  - Symbolic address becomes a **loadI** / **load** sequence (as in lab 1)
  - Formula becomes a more complex expression
    - → Formula may be an offset from some known point, a chain of one or more pointers, an indirect reference through a class definition, …

We need to deal with the issue of storage layout, but not today

Example — Emitting ILOC

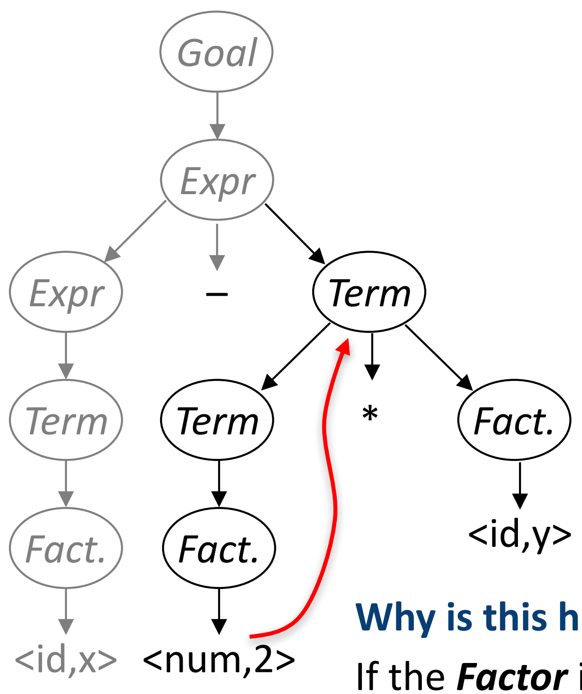
Stack	Input	Action
$	id – num * id	*shift*
$ id	– num * id	*reduce 10*
$ *Factor*	– num * id	*reduce 7*
$ *Term*	– num * id	*reduce 4*
$ *Expr*	– num * id	*shift*
$ *Expr* –	num * id	*shift*
$ *Expr* – num	* id	*reduce 9*
$ *Expr* – *Factor*	* id	*reduce 7*
$ *Expr* – *Term*	* id	*shift*
$ *Expr* – *Term* *	id	*shift*
$ *Expr* – *Term* * id		*reduce 10*
$ *Expr* – *Term* * *Factor*		*reduce 5*
$ *Expr* – *Term*		*reduce 3*
$ *Expr*		*accept*

What about using multI ?

- Could fold the "2" into the multiply by using multI
- Requires a non-local computation
  - Reach across productions to expose the difference between num & ident

```
loadAI   pr0, @x    ⇒ vr1
loadI    2          ⇒ vr2
loadAI   pr0, @y    ⇒ vr3
mult     vr2, vr3   ⇒ vr4
sub      vr1,vr4    ⇒ vr5
```

Example — Emitting ILOC



Syntax tree

Why is this hard to do with AHSDT?

If the ***Factor*** is an id, then the rules need to pass a register upward,

If the ***Factor*** is a num, then the rules need to return either a register or an immediate value, depending on whether it is the right or left child of the ***mult.***

The rules rapidly become context dependent in complex ways.

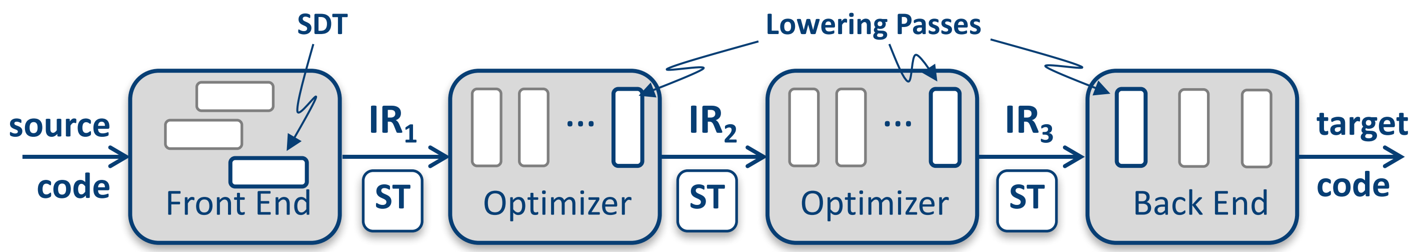
What about using multI ?

- Could fold the “2” into the multiply by using multI
- Requires a non-local computation
  - Reach across productions to expose the difference between num & ident
  - Difficult to describe in the limited vocabulary of AHSDT (see EaC2e § 8.4.1)

```
loadAI   pr0, @x    ⇒ vr1
loadI    2          ⇒ vr2
loadAI   pr0, @y    ⇒ vr3
mult     vr2, vr3   ⇒ vr4
sub      vr1,vr4    ⇒ vr5
```

Code Generation

In a modern, multi-IR compiler, code generation happens several times



- Generate an **IR** straight out of the parser
  - Might be an **AST**, might be some high-level (abstract) linear form
  - Almost always accompanied by a “symbol table” of some form
- Code passes through one or more “lowering” pass
  - Takes the code and decreases (“lowers”) the level of abstraction
  - Expand complex operations (e.g., **call** or **mvcl**), make control-flow explicit

The problems are, essentially, the same

The mechanisms are likely quite different

Mechanisms for Code Generation

Might
STOP

In different contexts, the compiler might use different techniques

- In an **LR(1)** parser, **SDT** might be the right tool
  - Generate a tree, a graph, or linear code
  - Actions driven by the syntax of the input code
- In a lowering pass, traversing the **IR** might be necessary
  - For a graphical **IR**: *A treewalk, or graph-walk, that produces the new **IR***
  - For a linear **IR**: *Walk through the **IR** in one direction or the other*
- In instruction selection, pattern matching is the tool of choice
  - The set of choices, particularly address modes, make this problem complex
  - Code is not subject to subsequent optimization

The context makes these tasks subtly different

- Early in compilation, generate code that will optimize well
- Late in compilation, generate code that will run well

A More Complex Example

Generating code for an if-then-else construct

Stmt	→	<u>IF</u> <u>LPAREN</u> *Expr* <u>RPAREN</u> <u>THEN</u> *WithElse* <u>ELSE</u> *WithElse*
	|	…
Expr	→	…
WithElse	→	…

- Control-flow constructs require branches and labels
- Need a schema for how to implement an if-then-else
  - Evaluate the expression
  - Based on its value, branch to the then part or the else part
  - After evaluating the appropriate part, branch to the next statement
    - → *We will call that the point in the code the "**exit**" to simplify talking about it*

We will assume that the grammar has Boolean & relational expressions

(Fig. 7.7, p 351 in EaC2e)

Boolean & Relational Expressions

First, we need to add boolean & relational expressions to the grammar

Boolean	$\rightarrow$	*Boolean* $\vee$ *AndTerm*	*Expr*	$\rightarrow$	*Expr* + *Term*
	|	*AndTerm*		|	*Expr* - *Term*
AndTerm	$\rightarrow$	*AndTerm* $\wedge$ *RelExpr*		|	*Term*
	|	*RelExpr*	*Term*	$\rightarrow$	*Term* × *Value*
RelExpr	$\rightarrow$	*RelExpr* < *Expr*		|	*Term* ÷ *Value*
	|	*RelExpr* ≤ *Expr*		|	*Value*
	|	*RelExpr* = *Expr*	*Value*	$\rightarrow$	! *Factor*
	|	*RelExpr* ≠ *Expr*		|	*Factor*
	|	*RelExpr* ≥ *Expr*	*Factor*	|	(*Expr*)
	|	*RelExpr* > *Expr*		|	number
	|	*Expr*		|	*Reference*

This allows
w < x < y < z

… where *Reference* derives a name, a subscripted name, a structure reference, a string reference, a function call, …

A More Complex Example

Generating code for an if-then-else construct

Stmt	→	IF LPAREN *Expr* RPAREN THEN *WithElse* ELSE *WithElse*
	|	…
Expr	→	…
WithElse	→	…

- Control-flow constructs require branches and labels
- To generate code with **SDT**, need to create & track the labels
    1. Create labels for "then part", "else part", and the "exit"
    2. Emit branch to appropriate part (then or else) after RPAREN
    3. Emit label for then part before first *WithElse*
    4. At end of *WithElse*, emit branch to the exit
    5. Emit label for else part before second *WithElse*
    6. At end of second *WithElse*, emit branch to exit
    7. Emit label for the exit on a **nop** after the second *WithElse*

That looks like a lot of work

A More Complex Example

Generating code for an if-then-else construct

We need a way to hang code snippets in the middle of the **RHS**

→ Use the trick from *GramSymbol* — an epsilon production

Stmt	→	IF LPAREN *Expr* RPAREN *CreateAndBranch* THEN *EmitThenLabel* *WithElse* *EmitExitJump* ELSE *EmitElseLabel* *WithElse* *EmitExitJump* *EmitExitLabel*
CreateAndBranch	→	ε
EmitThenLabel	→	ε
EmitExitJump	→	ε
EmitElseLabel	→	ε
EmitExitLabel	→	ε

A More Complex Example

Generating code for an if-then-else construct

- To generate code with **SDT**, need to create & track the labels
  1. Create labels for “then part”, “else part”, and “exit”
     - → *Need a structure to hold the three labels*
     - → *Generate three labels and push them onto the stack in CreateAndBranch action*
  2. Emit branch to appropriate part (then or else) after RPAREN
     - → *Emit the branch in CreateAndBranch action*
  3. Emit label for then part before first *WithElse*
     - → *Emit a labelled **nop** in EmitThenLabel action*
  4. At end of *WithElse*, emit branch to the exit
     - → *Emit a jump to the exit label in EmitExitLabel action*
  5. Emit label for else part before second *WithElse*
     - → *Emit a labelled **nop** in EmitElseLabel action*
  6. At end of second *WithElse*, emit branch to the exit
     - → *Handled by 4. above*
  7. Emit label for the exit on a **nop** after the second *WithElse*
     - → *Emit a labelled **nop** in EmitExitLabel action*

A More Complex Example

Bison supports this idea

- Allows code snippets between any two grammar symbols on the **RHS** of a production
- Generates the appropriate epsilon production, its reduction, and ties the action to this new reduction
  - Actions work in the name space of the production where they are written
  - Allows the notation to handle these code snippets in a natural way

Sample from the compiler for DEMO

```
ITE:  IF LP Bool RP
        {  $$ = new ITEHEAD ;
           $$->then_label = NextLabel;
           $$->else_label = NextLabel;
           $$->exit_label = NextLabel;
           emit_branch($3->reg,$$->then_label,
                           $$->else_label);
           emit_nop($$->then_label,"then part");
        }
        THEN WithElse
        {
           emit_jump($$->exit_label);
           emit_nop($$->else_label);
        }
        ELSE WithElse
        {
           emit_jump($$->exit_label);
           emit_nop($$->exit_label);
        }
```

Combines several items into one action

This does get tricky. Bison introduces the ε-productions, which can introduce new shift-reduce conflicts and changes the numbering of the $i macros. (See the Bison manual.)