TDT4205 Compilers

Exercise 3

Stian Hvatum (hvatum)
MTDT

February 21, 2012

PART 1 Theory

Task 1.1 Parsing

1.1.1 LL(k)

Even if LL(k) in theory can be extended with an $\rightarrow infinitly$ number of lookaheads, this will not resolve the problems with left recursion. LL(k) parses in a naive way, and if there exists a left recursion, like S $\rightarrow$ Ss $|\epsilon$, it will allways match the first S, and will spin generating more S's. This means we will continue beond the number of s's in the parsed text, and things will go bad.

Even if we could modify the LL(k) to recognize patterns using the lookaheads, and match for a spesific number of recursive calls, we would run into practical problems. For each lookahead-symbol extra we need, the parse table grows, and at the end it will become humongous. To create and use such a table is both space and time consuming, and we will never have neither space nor time to parse languages and grammars with arbitary use of left recursion, as it requires arbitary much space and time. Also, LL(k) needs the k to be defined, wich sets an upper bound for the number of lookaheads before we begin parsing. We may set the k to 10000000000, and hope that the language users will never exeed this number of recursions, but it will not be a *valid* parser for that language, as it can only handle a subset of the possible language constructs.

1.1.2 Left-recursive grammars

The left-recursive grammar:

F	$\rightarrow$	f I v A w S x
A	$\rightarrow$	P
P	$\rightarrow$	P I $\|\epsilon$
S	$\rightarrow$	S s | s
I	$\rightarrow$	i

The equivalent non-left-recursive grammer:

F	$\rightarrow$	f I v A w S x
A	$\rightarrow$	P
P	$\rightarrow$	I P $\|\epsilon$
S	$\rightarrow$	s S'
S'	$\rightarrow$	s S' $\|\epsilon$
I	$\rightarrow$	i

1.1.3 `FIRST` and `FOLLOW` & LL(1) Parsing table

`FIRST` and `FOLLOW`

Computing `FIRST`

- f is in FIRST(F), since f is the first symbol in production of F, and f is a terminal, and thereby FIRST(f) = f.
- i is in FIRST(I), since FIRST(I) = FIRST(i) = i
- i is in FIRST(P), since FIRST(P) = FIRST(I) = i
- ϵ is in FIRST(P), since P has a production P $\rightarrow \epsilon$
- i, ϵ is in FIRST(A), since FIRST(A) = FIRST(P) = i, ϵ
- s is in FIRST(S) since FIRST(S) = FIRST(s) = s
- s is in FIRST(S') since FIRST(S') = FIRST(s) = s
- ϵ is in FIRST(S'), since S' has a production S' $\rightarrow \epsilon$

Computing `FOLLOW`

- We start by adding $ to F, since F is our *start-symbol.*
- We see from F $\rightarrow$ f I v A w S x that w is in FOLLOW(A), x is in FOLLOW(S) and v is in FOLLOW(I)
- FOLLOW(P) = FOLLOW(A) since A $\rightarrow$ P.
- FOLLOW(S') = FOLLOW(S) since S $\rightarrow$ S'.

- FOLLOW(I) includes FIRST(P) except ϵ since P $\rightarrow$ I P $|\epsilon$, and rule 3, wich makes FOLLOW(I) = i, v

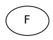
NT	FIRST	FOLLOW
F	f	$
A	i, ϵ	w
P	i, ϵ	w
S	s	x
S'	s, ϵ	x
I	i	i, v, w

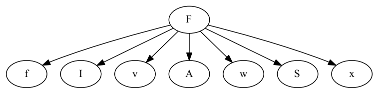
LL(1) Parsing table

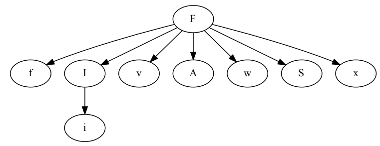
Followed page 224, Alg. 4.31
IF A $\rightarrow$ b AND FIRST(b) contains c THEN add A $\rightarrow$ b to M(A,c) and
IF FIRST(b) contains ϵ, THEN add A $\rightarrow$ ϵ to M(A,c) M(A,b) M(A,c)

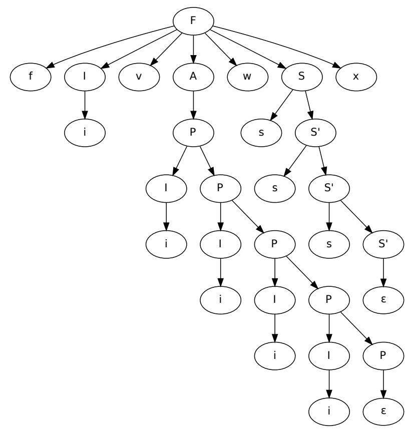
Non-Terminal	Input Symbol						
	f	i	v	w	s	x	$
F	F $\rightarrow$ f I v A w S x						
A		A $\rightarrow$ P		A $\rightarrow$ ϵ			
P		P $\rightarrow$ I P		P $\rightarrow$ ϵ			
S					S $\rightarrow$ sS'		
S'					S' $\rightarrow$ sS'	S' $\rightarrow$ ϵ	
I		I $\rightarrow$ i					

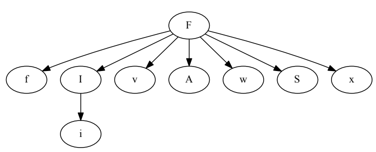
1.1.4 Parse tree for LL(1)











1.1.5 Buttom-up parsing

Symbol	Input	Action
\$	f i v i i i i w s s s s x \$	Shift f
\$ f	i v i i i i w s s s s x \$	Shift i
\$ f i	v i i i i w s s s s x \$	Reduce I $\rightarrow$ i
\$ f I	v i i i i w s s s s x \$	Shift v
\$ f I v	i i i i w s s s s x \$	Reduce P $\rightarrow \epsilon$
\$ f I v P	i i i i w s s s s x \$	Shift i
\$ f I v P i	i i i w s s s s x \$	Reduce I $\rightarrow$ i
\$ f I v P I	i i i w s s s s x \$	Reduce P I $\rightarrow$ P
\$ f I v P	i i i w s s s s x \$	Shift i
\$ f I v P i	i i w s s s s x \$	Reduce I $\rightarrow$ i
\$ f I v P I	i i w s s s s x \$	Reduce P I $\rightarrow$ P
\$ f I v P	i i w s s s s x \$	Shift i
\$ f I v P i	i w s s s s x \$	Reduce I $\rightarrow$ i
\$ f I v P I	i w s s s s x \$	Reduce P I $\rightarrow$ P
\$ f I v P	i w s s s s x \$	Shift i
\$ f I v P i	w s s s s x \$	Reduce I $\rightarrow$ i
\$ f I v P I	w s s s s x \$	Reduce P I $\rightarrow$ P
\$ f I v P	w s s s s x \$	Reduce A $\rightarrow$ P
\$ f I v A	w s s s s x \$	Reduce A $\rightarrow$ P
\$ f I v A	w s s s s x \$	Shift w
\$ f I v A w	s s s x \$	Shift s
\$ f I v A w s	s s x \$	Reduce S $\rightarrow$ s
\$ f I v A w S	s s x \$	Shift s
\$ f I v A w S s	s x \$	Reduce S $\rightarrow$ S s
\$ f I v A w S	s x \$	Shift s
\$ f I v A w S s	x \$	Reduce S $\rightarrow$ S s
\$ f I v A w S	x \$	Shift x
\$ f I v A w S x	\$	Reduce F $\rightarrow$ f I v A w S x
\$ F	\$	Accept

Task 1.2 Symbol tables

1.2.1 Data structure

Symbol tables are often represented as one or more hash tables. Hash tables provide fast lookup, so that we have rapid access to the information about the current symbol. We may use multiple hash tables to ease the parsing though different scopes of the parsed program, eg. one hash table for each scope, stored in a hierarchy.

1.2.2 Function pointers

When we allow function pointers in our language, we may hit several problems during compile time. If the symbol table is just like a symbol table for programing languages without function pointers, we cannot know if `dostuff` is a variable or a function, neither do we know how many arguments it take or if it returns anything. Since we cannot raise errors about things we don't know about, we must just hope that the program runs correct at run-time.

1.2.3 Extending the symbol table

If we add type (FUNC / VAR / FUNC_PTR), returns symbol (Y/N) and number of arguments to the symbol table, then we may raise errors during compile time if the user has mixed up types or number of arguments.

1.2.4 Structure of a symbol table entry

Symbol	**Mem loc**	**Type**	**Returns val**	**N args**
f	dostuff	FUNC_PTR	-	-
dostuff	`mem_loc`	FUNC	false	1

Where

- Symbol is the referance symbol.
- Mem loc is the location in memory where the function or the variable is located, or the referance symbol if it references an other symbol.
- Type is wherever this is a function, a variable or a function pointer.
- Returns val is wherever the function, if it is a function, has a return statement.
- N args is the number of arguments the function takes, or 0 if it is not a function.

Task 1.3 Syntax-Directed Translations

1.3.1 What is a SDD?

Syntax-directed definition is our grammar with added support for attributes. Attributes is any kind things we can store in memory, like numbers or strings. This is usefull when we need to add semantic actions to our parse tree, eg. create an AST, wich again is usefull for type checking and intermediate code generation.

1.3.2 L- and S-attributed SDD

What is the difference between L-attributed and S-attributed syntax- directed definitions? What does Bison support? Please explain.

L- and S-attributed SDD's are two classes of SDD's that do guarantee evaluation order and does not permit dependency graphs with cycle. S-attributed SDD's are

1.3.3 SDD and Grammars

E →	E + T	{}
E →	T	{}
T →	num , num	{}
T →	num	{}