Kompilator języka bazującego na C do języka asemblera ARMa

Adam Rzepka
Informatyka Stosowana
III rok
nr albumu: 219977

17 maja 2011

1 Wstęp

W dokumencie została zaprezentowana gramatyka, która będzie wykorzystana do generowania parsera podzbioru języka C, przy użyciu biblioteki cl-yacc (język Common Lisp). Gramatyka ta nie obsługuje słów kluczowych:

- struct,
- typedef,
- union,
- enum,
- auto,
- register,
- static,
- extern,
- const,
- volatile,

oraz operatorów:

- ?:
- ->
- .
- +=, -= itp.

Z uwagi na to, że preprocesor będzie zaimplementowany osobno, gramatyka nie uwzględnia również jego dyrektyw i makr. Projekt będzie rozwijany jako praca inżynierska, dlatego z czasem zostaną dodane brakujące elementy języka, jak również kilka nowych funkcjonalności.

2 Opis tokenów

LP	Nazwa	Opis
1	char	char
2	do	do
3	double	double
4	else	else
5	float	float
6	for	for
7	if	if
8	int	int
9	long	long
10	return	return
11	short	short
12	sizeof	sizeof
13	void	void
14	while	while
15	identifier	[A-Za-z_][A-Za-z0-9_]*
16	constant	[0-9]+[uUlL]?
17	constant	0[0-7]+[uUlL]?
18	constant	(0x|0X)[0-9A-Fa-f]+[uUlL]?
19	constant	[0-9]+.[0-9]*([eE][+-]?[0-9]+)?[fFlL]?
20	constant	[0-9]*.[0-9]+([eE][+-]?[0-9]+)?[fFlL]?
21	constant	[0-9]+[eE][+-]?[0-9]+)[fFlL]?
22	constant	L?'([^\\']|\\')+'
23	string	L?"([^\\"]—|\\")+"
24	>>	>>
25	<<	<<
26	++	++
27	–	–

28	\&\&	&&
29	||	||
30	<=	<=
31	>=	>=
32	==	==
33	! =	! =
34	\;	;
35	{	{
36	}	}
37	,	,
38	=	=
39	\(	(
40	\)	)
41	[	[
42	]	]
43	.	.
44	\&	&
45	!	!
46	~	~
47	-	-
48	+	+
49	*	*
50	/	/
51	%	%
52	<	<
53	>	>
54	^	^
55	|	|

3 Gramatyka

```
;; cl-yacc parser
(yacc:define-parser *c-parser*
  (:start-symbol file)
  (:terminals (double do else float for if int long return
               short sizeof void while identifier constant
               string << >> ++ -- \&\& \|\| <= >= == != \; { }
               \, = \( \) [ ] ! ~ - + * / % < > ^ \|))
  ;; Zdefiniowanie priorytetow i lacznosci operatorow
  ;; pozwala zredukowac ilosc produkcji
  (:precedence ((:left * / %) (:left + -) (:left << >>)
                (:left < > <= >=) (:left == !=) (:left &)
                (:left ^) (:left \|) (:left \&\&) (:left \|\|)
                (:right =) (:left \,) (:nonassoc if else)))

  (file
    (declaration \;)
    (file declaration \;)
    function
    (file function))

  (declaration
    (type var-init-list))

  (var-init-list
    (var-init-list \, pointer-declarator = initializer)
    (pointer-declarator = initializer)
    (var-init-list \, pointer-declarator)
    pointer-declarator)

  (pointer-declarator
    declarator
    (pointer declarator))

  (declarator
    identifier
    (\( declarator \))
    (declarator [ expression ])
    (declarator [ ] )
    (declarator \( param-list \)))
```

```
(pointer
  *
  (pointer *))

(initializer
  ({ initializer-list })
  expression)

(initializer-list
  (initializer-list \, initializer))

(function
  (type pointer-declarator \( param-list \) block))

(type
  double
  float
  int
  long
  short
  void)

(param-list
  (param-list \, declaration)
  (declaration))

(block
  ({ })
  ({ instruction-list })
  ({ declaration-list })
  ({ declaration-list instruction-list }))

(declaration-list
  declaration
  (declaration-list declaration))

(instruction-list
  (instruction-list instruction)
  instruction)

(instruction
  block
  (expression-instr)
  conditional
```

```
  loop)

(expression-instr
  \;
  (expression \;))

;; Pomimo, iz ponizsza produkcja wprowadza niejednoznacznosc,
;; jest ona dopuszczalna, dzieki zdefiniowaniu priorytetow
;; operatorow.
(expression
  cast-expression
  (expression * expression)
  (expression / expression)
  (expression % expression)
  (expression << expression)
  (expression >> expression)
  (expression > expression)
  (expression < expression)
  (expression >= expression)
  (expression <= expression)
  (expression == expression)
  (expression != expression)
  (expression & expression)
  (expression ^ expression)
  (expression \| expression)
  (expression \&\& expression)
  (expression \|\| expression)
  (unary-expression = expression)
  (expression \, expression))

(cast-expression
  unary-expression
  (\( type \) cast-expression))

(unary-expression
  postfix-expression
  (++ unary-expression)
  (-- unary-expression)
  (+ cast-expression)
  (- cast-expression)
  (* cast-expression)
  (& cast-expression)
  (! cast-expression)
  (~ cast-expression)
```

```
  (sizeof unary-expressiion)
  (sizeof \( lvalue \)))

(postfix-expression
  (postfix-expression \( arument-list \))
  (postfix-expression \( \))
  (postfix-expression [ expression ])
  (postfix-expression ++)
  (postfix-expression --)
  (highest-expression))

(argument-list
  expression
  (argument-list \, expression))

(highest-expression
  identifier
  constant
  string-literal
  (\( expression \)))

(conditional
  (if \( expression \) instruction else instruction)
  (if \( expression \) instruction))

(repeat
  (for \( expression-instr expression-instr expression \) instruction)
  (for \( expression-instr expression-instr \) instruction)
  (while \( expression \) instruction)
  (do instruction while \( expression \))))
```